Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Issue Tracking Ecosystems
Context and Best Practices

Dissertation for the doctoral degree **(Dr. rer. nat.)**
at the Faculty of Mathematics, Informatics, and Natural Sciences
Department of Informatics
University of Hamburg
Hamburg, Germany

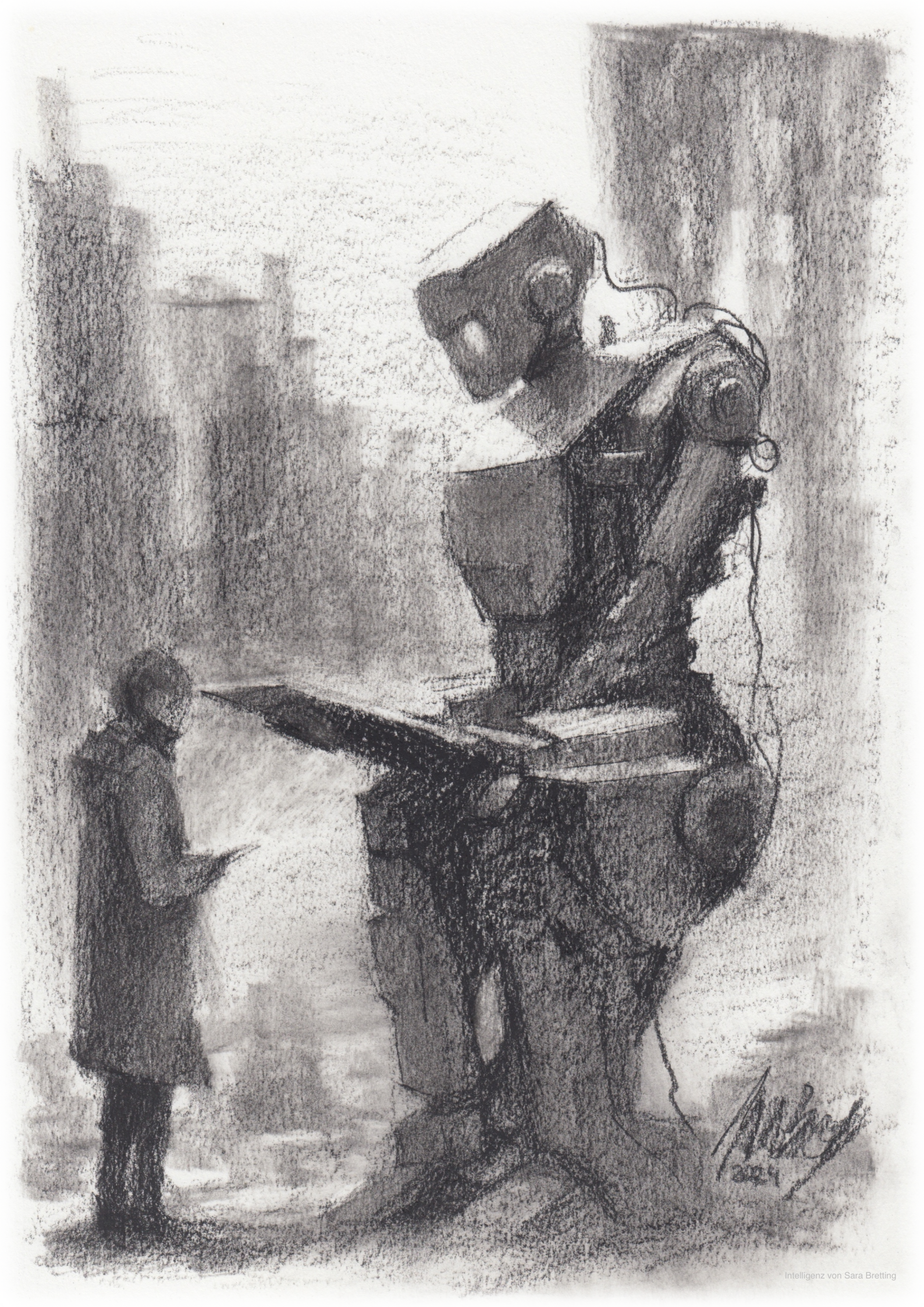
submitted by

Lloyd Montgomery

from White Rock,
British Columbia, Canada

Hamburg, 2025

Oral Defence Date	April 11th, 2025
Examination Commission	Prof. Dr. Thomas Ludwig (Chair) Prof. Dr. Peter Kling (Vice Chair)
Dissertation Evaluators	1. Prof. Dr. Walid Maalej 2. Prof. Dr. Andreas Vogelsang 3. Ass. Prof. Dr. Eray Tüzün



To my wife, Lisa.

Acknowledgements

The first person I need to acknowledge is my wife, Lisa. She provided me with both the encouragement to push through difficult times, and the kindness to tell me when to take a break. Most importantly, she taught me the value of living a full life, a life beyond the PhD.

I am deeply grateful for the opportunities afforded to me by my Doktorvater, Walid Maalej. Walid's acclaim in the research community brought me to his research group in Germany. His hard work and dedication provided me a position in an EU-funded project that enabled me to travel Europe and collaborate with internationally renown researchers. His guidance elevated me into the research community as a respected scientist, reviewer, and peer. Lastly, his patience and feedback supported my journey towards achieving my PhD. I am forever grateful.

I'd like to thank my family for their continued support throughout my academic journey. My sister, Alexis, has provided me with a lifetime of sibling joy and connection. My mother, Jennifer, has been patient and supportive through my growth as an individual. My father, Dale, has always been there for me providing both support and entertainment. My grandmother, Julie, has always been there as a steady, reliable source of wisdom and delicious food. My stepfather, Mike, has provided stability to our second family and is always there for my mother. I'd also like to acknowledge my new family who have accepted and embraced me as one of their own. Thank you to Christoph, Anja, Lea, and Mathis for welcoming me into their home. Without their feedback and support, this journey would have been much harder.

My friends have brought me untold amounts of joy, acceptance, patience, and support. Volodymyr has been a good friend ever since I first started my PhD, and has provided much needed entertainment. Davide was always there when an event was to be had, a real friend you could count on. Guy is someone I can always go to for brutally honest feedback, which is something we all need every now and then. Dorothea has wonderfully refreshing perspectives on both academia and life in general. Thomas and Ela are two beautifully kind souls who brought me into their life and provided me with my first home in Germany. Svenja warmed my heart, and together we formed many lasting memories together. In recent years, I am grateful to have many new and meaning connections in my life, including Tim, Aref, and Ariane. Without such kind and giving people around me, this journey would have been much more difficult.

The MAST research group has been my home away from home, filled with people I deeply respect and have formed many strong memories with. Davide, Daniel, Christoph, and Volodymyr have been great colleagues since I first started my PhD. They have challenged my understanding of the world, improved my work, and helped shape my academic career. Marlo, Clara, Yen, and Abir brought many interesting conversations that improved my perspective on many topics. Christian, Tim Puhlfürß, Tim Pietz, and Aref have all been fantastic colleagues to get to know over the past few years. Jakob, Pavel, Michael, and Amna have all brought meaningful outside perspectives into the MAST group. I also want to thank Farnaz, Alexander, and Zijad for their feedback and support on various academic topics. I'd like to thank those who gave me feedback on my thesis: Tim, Abir, Volodymyr, Pavel, Christian, and Michael. I'd also like to thank the University of Hamburg for hosting my PhD, and providing a platform for me to teach and mentor many intelligent and talented individuals.

The Software Engineering research community has many characteristics—many positive and some negative. What is most positive about this community, is the people who stand out as champions of strong scientific principles, act as meaningful mentors, and offer paradigm-shifting ideals—none of which are extrinsically rewarded by the academic system. In this category of people, I'd like to give a special thank you to Daniel Mendez, Julian Frattini, Marvin Wyrich, Davide Fucci, Alessio Ferrari, Andreas Vogelsang, Margaret-Anne Storey, and Alexander Serebrenik. Daniel Mendez understands and applies the true value of empathy within our community, and he knows how to support those around him. If was his words of encouragement, and perspective-altering discussions about the Software Engineering community, that kept me pushing through my PhD. Julian Frattini and Marvin Wyrich are beacons of excellence in a system of mediocracy; they know where meaningful change is needed, and work to achieve it. Both Julian and Marvin have given me much hope in my outlook on the Software Engineering research community: that there exists a future generation of scientists who will make a meaningful difference. Davide Fucci understands the value of community, and works vigorously to keep people connected, invested, and producing the best output they are capable of. He has had a positive impact on my experience in this community by showcasing just how powerful bringing people together can be. Alessio Ferrari and Andreas Vogelsang have been champions for young scientists looking to join, integrate, and excel in our community. While I have not had the privilege of working directly with either of them, they have both shown me kindness as a young member of the community, encouraging me to continue the academic journey. Margaret-Anne Storey and Alexander Serebrenik are academics who have connected with me along my PhD journey, offering their time, their attention, and their advice. Each of them is a highly respected prominent member of the community in their respective research areas, giving keynotes at top-tier Software Engineering venues, and yet they immediately drop everything when they sense that you need them. I received words of wisdom from both of them at critical points in my PhD journey, and I am very thankful for their advice.

The PhD journey is like competing in the Olympics: you showcase feats of personal intelligence, skill, and endurance, but this is only possible because of the team of coaches, peers, family, and friends who enable them.

Thank you team.

Abstract

Issue Tracking Systems (ITSs), such as GitHub and Jira, are popular tools that support Software Engineering (SE) organisations through the management of "issues", which represent different SE artefacts such as requirements, development tasks, and maintenance items. ITSs also support internal linking between issues, and external linking to other tools and information sources. This provides SE organisations key forms of documentation, including forwards and backwards traceability (e.g., Feature Requests linked to sprint releases and code commits linked to Bug Reports). An Issue Tracking Ecosystem (ITE) is the aggregate of the central ITS and the related SE artefacts, stakeholders, and processes—with an emphasis on how these contextual factors interact with the ITS. The quality of ITEs is central to the success of these organisations and their software products. There are challenges, however, within ITEs, including complex networks of interlinked artefacts and diverse workflows. While ITSs have been the subject of study in SE research for decades, ITEs as a whole need further exploration.

In this thesis, I undertake the challenge of understanding ITEs at a broader level, addressing these questions regarding complexity and diversity. I interviewed practitioners and performed archival analysis on a diverse set of ITSs. These analyses revealed the context-dependent nature of ITE problems, highlighting the need for context-specific ITE research. While previous work has produced many solutions to specific ITS problems, these solutions are not consistently framed in a context-rich and comparable way, leading to a desire for more aligned solutions across research and practice. To address this emergent information and lack of alignment, I created the Best Practice Ontology for ITEs. Using this ontology, I curated a catalogue of Best Practices from existing literature, including Timely Severe Issue Resolution, Bug-to-Commit Linking, and Avoid Zombie Bugs. I also collected and created algorithms to automatically detect violations to these Best Practices. Finally, I proposed and evaluated tooling solutions that describe how to integrate these Best Practices into existing development environments.

The findings from this thesis enable a structured approach to improving the quality of ITEs. The Best Practice ontology, catalogue, and algorithms are contributions to researchers interested in understanding and improving ITEs. In practice, the context-aware catalogue and algorithms can be used to identify key areas for improvement, and to automate organisational processes such as maintaining a meaningful backlog and ensuring the completeness of issues.

Zusammenfassung

Issue Tracking Systems (ITSs) wie GitHub und Jira sind beliebte Tools, die Software Engineering (SE) Organisationen durch die Verwaltung von „Issues“ unterstützen, die verschiedene SE-Artefakte wie Anforderungen, Entwicklungsaufgaben und Wartungselemente darstellen. ITS unterstützen auch die interne Verknüpfung von Issues und die externe Verknüpfung mit anderen Tools und Informationsquellen. Dies bietet SE-Organisationen wichtige Formen der Dokumentation, einschließlich vorwärts und rückwärts gerichteter Rückverfolgbarkeit (z. B. Verknüpfung von Feature Requests mit Sprint Releases und Code Commits mit Bug Reports). Ein Issue Tracking Ecosystem (ITE) ist die Gesamtheit des zentralen ITS und der damit verbundenen SE-Artefakte, Stakeholder und Prozesse—mit dem Schwerpunkt darauf, wie diese kontextuellen Faktoren mit dem ITS interagieren. Die Qualität der ITEs ist entscheidend für den Erfolg dieser Organisationen und ihrer Softwareprodukte. Es gibt jedoch Herausforderungen innerhalb von ITEs, einschließlich komplexer Netzwerke miteinander verbundener Artefakte und unterschiedlicher Arbeitsabläufe. Während ITS seit Jahrzehnten Gegenstand der SE-Forschung sind, müssen ITEs als Ganzes weiter erforscht werden.

In dieser Arbeit stelle ich mich der Herausforderung, ITEs auf einer breiteren Ebene zu verstehen und diese Fragen hinsichtlich Komplexität und Vielfalt zu beantworten. Ich befragte Praktiker und führte eine Archivanalyse einer Vielzahl von ITS durch. Diese Analysen haben die kontextabhängige Natur von ITS-Problemen und -Lösungen aufgezeigt und den Bedarf an kontextspezifischer ITS-Forschung verdeutlicht. Während frühere Arbeiten vielfältige Analysen und Lösungen für spezifische ITS-Probleme hervorgebracht haben, sind diese Lösungen nicht durchgängig kontextabhängig und vergleichbar, was zu dem Wunsch nach besser abgestimmten Lösungen in Forschung und Praxis führt. Um dieses Informationsdefizit und den Mangel an Abstimmung zu beheben, habe ich eine ITE-Best-Practice-Ontologie erstellt. Anhand dieser Ontologie habe ich einen Katalog bewährter Verfahren aus der vorhandenen Literatur zusammengestellt. Anschließend habe ich Tooling-Lösungen vorgeschlagen und bewertet, die beschreiben, wie diese bewährten Verfahren in bestehende Umgebungen integriert werden können.

Die Erkenntnisse dieser Arbeit ermöglichen einen strukturierten Ansatz zur Verbesserung der ITE-Qualität. Die Ontologie, der Katalog und die Algorithmen sind Beiträge für Forscher, die sich für das ITE-Verständnis und die -Verbesserung interessieren. In der Praxis können der kontextbezogene Katalog und die Algorithmen genutzt werden, um Schlüsselbereiche für Verbesserungen zu identifizieren und organisatorische Prozesse zu automatisieren.

Contents

Part I.

Foundation Staging

Chapter 1.

Introduction

> The purpose of software engineering is to control complexity, not to create it.
>
> Dr. Pamela Zave

1.1. Problem Statement

Software Engineering (SE) is the process of planning, developing, testing, and maintaining software [213]. SE organisations conduct various forms of SE, ranging from more traditional models involving heavy processes and strict stakeholder hierarchy [146], to more open models of light-weight processes and open stakeholder communication [56]. Despite the rich and diverse landscape of SE processes, it is rare to find an SE organisation today that does not use some form of Issue Tracking System (ITS) to support their SE processes. An ITS is a software tool that structures information in the form of independent "issues", which are individual units of work to be conducted within organisations. For many teams, ITSs are the main place to collect and manage many types of SE artefacts, including requirements [233], development [139], and maintenance [27, 250]. This is the case, for example, with many Open-Source Software (OSS) projects hosted on GitHub such as Mastodon[1] and ReactJS.[2] An Issue Tracking Ecosystem (ITE) is the ITS, and all surrounding contextual factors that interface with the ITS. This involves the SE processes and stakeholders that interface with the ITS. The quality of an ITE refers to the extent to which it supports the organisation, including the features offered by the central ITS and the organisationally defined processes around it.

The benefit of ITEs for SE organisations are manifold, including communication and collaboration [26], workflow management, and user support [165]. ITEs have emerged as a central place for communication and collaboration, particularly among OSS communities [26, 157]. Git has also been tightly linked to other SE processes using ITSs, for example, GitHub has automatic linking of issues and Git commits using a custom ID system. ITSs also unmask the complex workflows within a SE organisation, encouraging more involvement from a diverse set

[1] `https://github.com/mastodon/mastodon`
[2] `https://github.com/facebook/react`

of stakeholders. For example, software users can see the status of Feature Requests. User support has also been enhanced by modern ITSs in that Bug Reports can be public and therefore shared among users. Bug reports can also be linked directly to SE artefacts such as user stories or release plans. Overall, the benefits of ITSs have lead to widespread adoption in industry [1, 46, 55] and many empirical studies seeking to understand these tools [27, 94, 98, 100, 250].

While central to modern SE, ITEs are notoriously difficult to navigate. They have captured the attention of SE research for over three decades, producing thousands of empirical findings across different research communities such as ICSE[3], FSE[4], RE[5], EMSE[6], IST[7], and JSS.[8] The diversity of ITSs also speaks to their inherent complexity: new ITSs continue to emerge as each attempts to simplify and solve the problems of the previous ones. While a few are well-known in SE, such as Jira [111], GitHub [83], GitLab [84], and Bugzilla [38], Wikipedia lists over 30 established ITSs,[9] in addition to the various other ITS-like tools that are not on the list such as asana [13], Smartsheet [212], and Monday [158]. Jira is the most popular ITS in industry [1, 46, 55]. In pop culture, Jira is repeatedly the target of jokes regarding its complexity and difficulty of use [113–115]. Making fun of Jira has become such a normalised online behaviour, that there is even a website dedicated entirely to showcasing negative quotes and opinions about Jira [2]. Research into ITSs has revealed some of this complexity (e.g., divergent expectations and difficulty with tooling [27, 80, 250]), but there is still much to be understood. **Gap 1**: *Our understanding of practitioner challenges with ITEs is limited.*

ITSs manage many SE artefacts and activities, including requirements, development, and maintenance. It is known, to some extent, that ITSs contain artefacts beyond just Bug Reports; however, this has not been studied in-depth. Studies have revealed the ITSs support "just-in-time" Requirements Engineering (RE) [56], similar to requirements processes conducted during "Agile SE" [98]. Studies have also investigated the user support aspect of ITSs [161]. However, the line between requirements, development, maintenance, and user support is blurred within ITSs [233], perhaps by design, or perhaps as a natural consequence of the needs of modern SE. ITSs are also a central tool for Agile SE, where iteration and evolution are key. Research investigating evolution within ITSs is rare [98], and there are still open questions regarding information and evolution within ITSs. **Gap 2**: *We are missing a holistic understanding of which artefacts and activities exist within ITSs, and what information exists and evolves.*

ITSs are highly customisable tools, offering personalisation through configuration and automation. This customisation is a major benefit of ITSs, but it also leads to complexity. How to use and configure parts of ITSs has also been extensively researched [27, 250]. However,

[3] `https://conf.researchr.org/series/icse`
[4] `https://conf.researchr.org/series/fse`
[5] `https://conf.researchr.org/series/RE`
[6] `https://link.springer.com/journal/10664`
[7] `https://www.sciencedirect.com/journal/information-and-software-technology`
[8] `https://www.sciencedirect.com/journal/journal-of-systems-and-software`
[9] `https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems`

these studies have focused on a specific issue type [27, 98, 99, 250] or process [56, 132, 140, 153, 215, 225, 228] within ITSs, and not the entire ITE. Additionally, existing recommendations are often contextless, offered as "fix all" solutions aimed at a general environment. **Gap 3**: *Context-dependent ITE recommendations for practitioners are understudied, including a structured way to document recommendations for future research, and communicate this to practitioners.*

1.2. Objectives and Contributions

The primary direction of this thesis is to *investigate, describe, and improve the quality of ITEs in both research and practice.* This includes four **objectives**, listed here:

O1 Specify and derive concepts essential to understanding and working with ITEs.

O2 Obtain an empirically grounded understanding of ITEs in practice.

O3 Improve the state of ITEs through theoretical and practical solutions.

O4 Recommend future directions for ITEs research and application in practice.

These four objectives form the structure of this thesis: Part I: Foundation Staging (O1), Part II: Problem Investigation (O2), Part III: Solution Construction (O3), and Part IV: Outlook Recommendation (O4). Within each part, there are several chapters, each with their own research questions that further explore and address the objectives. Each research question is addressed using empirical methods that produce rigorous findings. Across these objectives, research questions, and their associated findings, nine central **contributions** emerged:

Foundation Staging (O1)

C1 "Issue Tracking Ecosystem (ITE)" as a fundamental concept.

Problem Investigation (O2)

C2 A grounded understanding of the problems practitioners face within ITEs.

C3 A dataset of 16 Jira repositories with 2.7 million issues and 30 million issues.

C4 A data-driven characterisation of ITS artefacts, activities, information, and evolution.

C5 The cross-study finding that "context is key" for ITE problems and solutions.

Solution Construction (O3)

C6 An ontology for Best Practices for ITEs.

C7 A catalogue of Best Practices for ITEs.

C8 Algorithms to support the automatic detection of violations to the Best Practices.

Outlook Recommendation (O4)

C9 Tooling recommendations to support Best Practices for ITE in industry.

1.3. Scope

The scope of this thesis is the quality of ITEs. The scope does not include the quality of SE processes in general, the quality of software, or the general quality of artefacts involved in the SE lifecycle. An ITE involves factors including SE processes; however, this thesis focuses only on constructs that are tightly coupled with ITSs. For example, the study of customer involvement in Agile processes is outside the scope of this work, but the study of sprint lengths is inside the scope of the work because ITSs manage and are directly affected by sprint lengths. However, this thesis is only concerned with sprint lengths (and similarly tightly coupled processes) to the extent that ITSs can impact and manage them. In other words, this thesis is not concerned with what sprint lengths are best for Agile, but rather how an ITE can support and guide the management of sprint lengths as a feature of ITSs. This applies to all SE processes discussed in this thesis that are tightly coupled with ITSs and ITEs.

1.4. Outline

The thesis is structured in four main parts: Foundation Staging, Problem Investigation, Solution Construction, and Outlook Recommendation. Figure 1.1 provides a visual overview of the thesis structure, from research gaps, through parts and chapters, and then finally the contributions. Here is a textual overview of the parts, with a description for each chapter.

Part I. Foundation Staging

Ch 1. Introduction Overview of the problem, objectives, contributions, and scope.

Ch 2. Background Description of key concepts in this thesis.

Part II. Problem Investigation

Ch 3. Challenges Investigation of ITE problems in industry through interviews with 26 SE practitioners who utilise ITSs regularly.

Ch 4. Artefacts & Activities Investigation of SE artefacts and activities in ITSs through archival data analysis of 16 Jira ITSs consisting of 2.7 million issues.

Ch 5. Information & Evolution Investigation of information within ITSs and evolution of that information through archival analysis of 13 Jira ITSs consisting 1.3 million issues and 13 million evolutions.

Part III. Solution Construction

Ch 6. Ontology Formalisation of ITE Best Practices through the empirical construction of an ontology designed to structure ITE quality research and offer an approachable format for practitioners.

Ch 7. Catalogue Consolidation of ITE quality research into a catalogue of ITE Best Practices to offer a starting point for both researchers and practitioners.

Ch 8. Algorithms Creation and replication of algorithms to automatically detection violations of ITE Best Practices in the catalogue.

Part IV. Outlook Recommendation

Ch 9. Tooling Proposal of tooling features central to the creation of a recommender system for ITE Best Practices in practice.

Ch 10. Conclusion Summarising the thesis.

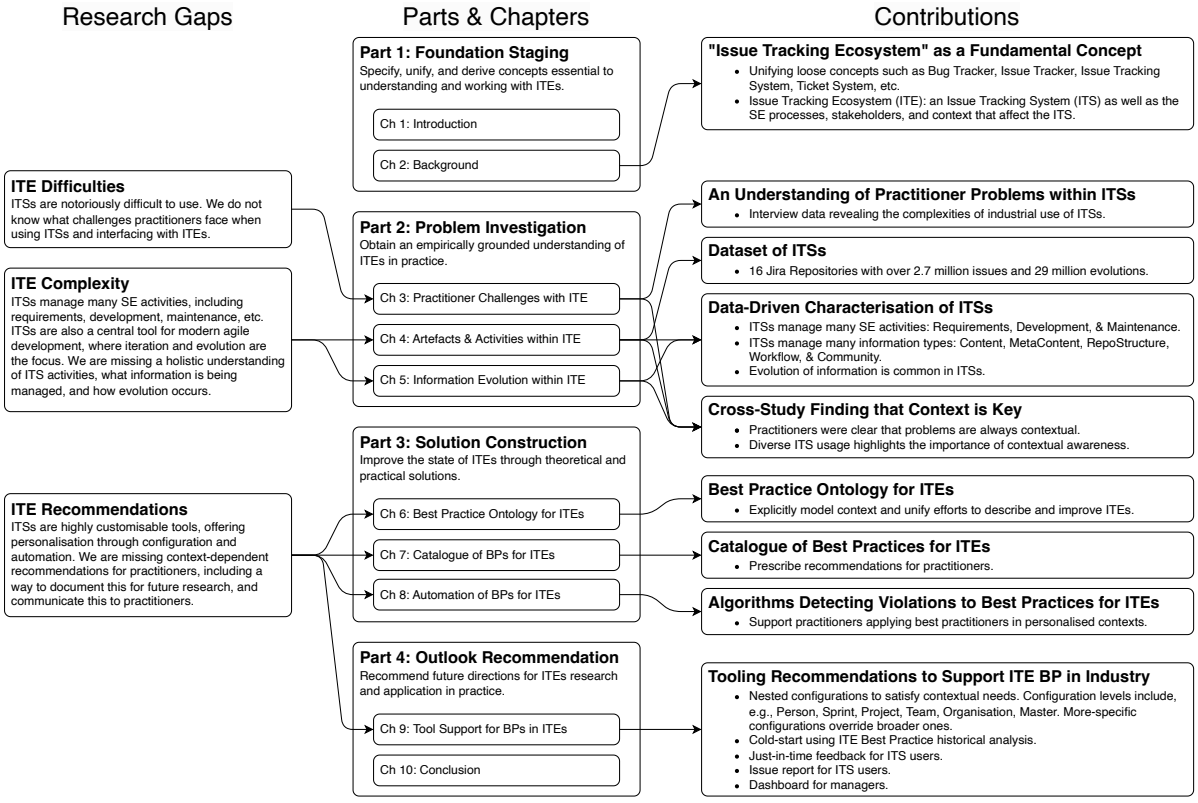


Figure 1.1.: Thesis overview: research gaps, parts, chapters, and contributions.

Chapter 2.

Background

> I would rather have questions that can't be answered than answers that can't be questioned.
>
> Richard Feynman

In this chapter, I outline the fundamental concepts used in this thesis. The concepts in this thesis are all within the scope of SE and concern the task of improving SE processes. In particular, this thesis describes ITSs (such as Jira [111], GitHub [83], and GitLab [84]). While ITSs originally stored primarily maintenance items (such as Bug Reports), they have grown to support most SE processes, including RE and development. The topic of maintenance in ITSs has received much research attention in the past two decades [27, 250], while the topic of requirements has not received nearly as much [98, 100, 233]. Accordingly, this chapter describes RE and how Agile practices have come to use and benefit from ITSs. Finally, this chapter discusses empirical methods, and how they are applied within this thesis.

Publications. This chapter contains portions of my collaborative publication, some copied verbatim [163, 165].

2.1. Requirements Engineering

RE is the process of gathering, documenting, verifying, and managing stakeholder needs, system constraints, and environmental factors for a system (often software) to be implemented. RE involves five fundamental phases: elicitation, analysis, specification, validation and verification, and management [146, 183, 224].[1] RE has a number of different forms and interpretations, including traditional, agile, and change-based [146, 149]. Each form shapes the interpretations of the definitions, processes, artefacts, outcomes, and research involved. This section outlines these three forms in sufficient detail to understand their similarities and differences, and discusses which forms are relevant to this thesis in which way.

There are five phases of RE [146, 224]. *Elicitation* is the process of gathering requirements

[1] The exact number of phases and the exact names used varies across sources, but they agree on the underlying concepts of these phases.

for a system. Traditionally, this has been done through direct contact with important stakeholders, but has evolved into a much more complex and comprehensive ecosystem including data- or user-driven RE [93, 144, 145, 174]. *Analysis* is the process of understanding the requirements and achieving consensus through modelling and discussions with various stakeholder groups. *Specification* is the process of documenting the requirements in some medium (textual or modelled). *Validation* is the process of checking if the system requirements—as documented—represent the system the stakeholders really want and need (e.g. "are we building the right system"). *Verification* is the process of checking if the system itself is correct, via some well-defined metrics such as completeness and compliance (e.g. "are we building the system right") [11]. Finally, *management* is the process of working with, updating, and maintaining the requirements throughout the duration of the project.

2.1.1. Traditional RE

Traditional Requirements Engineering is the first conceptualisation and formalisation of RE, often called "Waterfall RE" [121, 128, 181]. The metaphor of "Waterfall" comes from the clear and well-defined phases that Traditional RE is based on, whereby passing from one phase of RE to the next is a one-way process (you cannot go back up the waterfall). This form of RE has the aim of creating upfront, well-structured, and complete documents representing the functional and non-functional requirements (and constraints) for a system [52]. These documents include a Problem Statement, a Software Requirements Specification (SRS), and a Software Design Document (SDD). The Problem Statement outlines the gap between the current and desired system, often framed as a distinct set of "problems" to be addressed. In some forms, a Problem Statement can contain much of what is included in a full SRS, including functional and non-functional requirements, but in a shorter, non-complete, and high-level summary form. The SRS fully describes the requirements for a system, communicated through a description of the existing system and surrounding environment, a list of functional and non-functional requirements, use cases, and a glossary of domain-specific terms. An SRS can be hundreds of pages long. The final document found in the Traditional RE process is the SDD. The SDD crosses the boundary from problem space into solution space, detailing aspects of the system to be built. An SDD contains low-level descriptions of the system and object design, including class and architectural diagrams, as well as aspects of interface design. In addition to the documentation created for the system, Traditional RE can also involve pre-contract documents such as a Request for Quotes, Request for Proposals, and Business Proposals. I outline these documents used in Traditional RE to compare with those of the following RE type. I will not be discussing these documents further in this thesis.

2.1.2. Agile RE

Agile Requirements Engineering (also referred to as "Agile SE") is the response to the major drawbacks found in Traditional RE, primarily the long, gated, non-iterative phases. The Agile Manifesto [70], released in February 2001 by 17 prominent SE researchers, consultants, and industry leaders, outlines a set of values for Agile SE:

Individuals and interactions over processes and tools.
Working software over comprehensive documentation.
Customer collaboration over contract negotiation.
Responding to change over following a plan.

The modern conceptualisation of "Agile" involves adopting one of a few models, including Kanban, Scrum, Lean, Extreme Programming, and Feature-Driven Development [5]. In principle, Agile SE is simply the application of iterative feedback cycles to the primary phases of SE, mainly RE, software development, and system maintenance. One common form of Agile SE is Scrum. The Scrum process involves small co-located teams working tightly together in small development "sprints" of 1–4 weeks. A Scrum team is organised and managed by a combination of the roles Product Owner and Scrum Master. The Scrum Master is a meta-role responsible for managing and aligning team practice with the Scrum framework. The Product Owner is the most requirements-centric role, as they are concerned with the needs of their customers, users, and stakeholders, and guiding the team to create a better software product for them.

In Scrum, a popular form of requirements documentation is called a "User Story". User Stories are one way to capture the needs and desires of software users. Each User Story captures a small, self-contained unit of development work framed from the perspective of one actual user, conceptualised into a user group. User Stories are traditionally written with the form "As a [user persona], I want to [action], so that I [can accomplish this goal]." Since User Stories are small and self-contained, there is no inherent way to build larger structures. For this, agile teams use another artefact type called the "Epic", which is a high-level goal for software development. Epics have their own description, and a group of User Stories contained within them. Epics can also be used to form larger structures by grouping other Epics within them. These Epics and User Stories are collected in an ITS in a list called the Product Backlog. The Product Backlog doesn't have a limit to the number of Epics and User Stories contained within it, but it is the responsibility of the Product Owner to keep it clean and useful. Product Backlogs are often stored within ITSs along with other SE artefacts.

2.1.3. Change-Based RE

Change-Based Requirements Engineering is an emergent form of RE focused on product maintenance and evolution, and is conducted primarily within ITSs [149]. While Traditional RE and Agile Requirements Engineering are both methodologies designed for *product creation*,

Change-Based RE has emerged as an important methodology for long-lived software systems where *product maintenance and evolution* are the focus. Change-Based RE utilises ITSs as a central hub where maintenance tasks (such as Bug Reports) are collected and prioritised. The current user base can submit Feature Requests to be considered for integration into the evolving product vision. Agile Requirements Engineering and Change-Based RE are often used together by Agile teams who use an ITS to manage their SE processes.

The importance of Change-Based RE, as an additional RE methodology beyond Agile, is that Change-Based RE is agnostic to how the original software product was formed. Long-lived software systems can adopt a Change-Based RE process as a sustainable way to keep a software product updated and relevant over time. OSS teams use Change-Based RE through a publicly available ITS platform such as GitHub. This allows them to keep the product alive through contributions from their active user base. This also allows them to maintain a useable product as software standards change and new hardware is released. Finally, it also allows OSS products to stay relevant through community suggestions for new features and the community effort of developing desired features. This applies to both Agile products and other products, as long as their maintainers desire this Change-Based RE paradigm for their software product.

2.1.4. Requirements Quality

The quality of requirements refers to the individual characteristics of requirements that both lead to a successful and cost-effective system, and solve the user's needs [48, 163]. Requirements quality has been the topic of research for many decades, including the seminal 1994 Standish Chaos Report describing the perceived importance of requirements on software project success [88]. This report outlines project success factors such as "clear statement of requirements" and "clear vision and objectives", as well as project failure factors such as "incomplete requirements". These factors are forms of quality for requirements, often called "quality factors" or "quality attributes". In 2007, Kamata and Tamai empirically validated the claims of the Standish Chaos Report through an investigation of 32 industrial software projects focusing on the impact of RE on quality, time, and cost calculations [120]. Their research found that projects with an acceptable level of quality across all sections of their requirements were within the time and cost budgets. These findings have also been confirmed more recently [150, 237].

The quality of requirements is closely related to the quality of ITEs. Requirements are often managed *within* ITEs, and are central to the processes therein. In the case of requirements issues in ITSs, *requirements quality* applies rather directly. Other artefact types in ITSs (such as development or maintenance issues) are similar to requirements because all issues are actions that the maintainers should perform. For the broader ITE around the ITS, requirements quality attributes such as "completeness", "correctness", and "consistency" can apply to many of the processes being conducted. Requirements quality is not an all-encompassing concept, but there is considerable overlap between requirements quality factors and the quality of ITEs.

2.2. Issue Tracking Systems and Ecosystems

An ITS is a tool used by SE organisations to manage the development and maintenance of software. Each *issue* in an ITS represents a stand-alone unit of work to be performed as part of the project, or a problem to be solved. The *issue type* represents the main purpose of the issue and the work it comprises: for instance, a "Bug Report" to be fixed or a "User Story" to be implemented. Within an ITS, issues are assigned to *projects*, which are mutually exclusive bins that structure the work and teams within an organisation. Each project usually has certain goals (e.g. the development of a certain tool), involves certain stakeholders, and follows some shared (explicit or tacit) practices and workflows. For example, "Hadoop Common" is an Apache project and "JBoss Application Server" is a RedHat project. Each issue is composed of several issue *fields*, where each field has a name and a value. The fields represent the information relevant for the issue management: including the understanding, planning, and resolution of the issues [247]. Common examples of fields include Summary, Description, Priority, and Comments. When an issue is created, basic information such as its summary and description are stored. Subsequently, the issue evolves until it gets resolved, meaning that the values of different fields are updated. Each of these *issue evolutions* has a timestamp, an author, the field name, as well as the value before and after.

2.2.1. Benefits of Issue Tracking Systems

ITSs are ubiquitous in the SE community. When learning about the workflow of a SE organisation, it is inevitable that one must ask about which ITS they are using. They could be using a simplified workflow with Trello [230], a long-lived tool such as Trac [229], or embracing the full complexity of Jira [111]. Regardless, ITSs clearly play an important role in SE organisations. Looking closely at the features and fields offered by ITSs, there are more specific reasons ITSs are so popular and important. One such reason is that OSS communities use them as a *central tool for communication and collaboration* [26, 157]. The central tool for OSS communities was once mailing lists, where each email thread represented a single discussion about a bug, feature, or wider topic. Now, those discussions are held in the comments section of issues, further enhanced by the other features offered by issues trackers such as labelling and linking [31, 138–140]. Another reason for their popularity is that modern ITSs are often used as *an enhancement to traditional versioning systems such as Git.* While Git captures most of what is necessary when it comes to versioning software development efforts, it rarely captures or links to the other SE artefacts such as requirements or Bug Reports. ITSs enable a rich enhancement through traceability between Git commits and the rationale behind them.

From an RE perspective, ITSs have created a direct avenue for *just-in-time requirements* [56] from diverse stakeholder groups. While there is merit in traditional RE processes in certain contexts—which often have a top-down approach to requirements planning—ITSs enable just-in-time requirements where individual needs and specific requests are combined with community

involvement ("Crowed RE" [87]). For companies interested in understanding "the crowd", traditional RE requires user studies, surveys, or interviews. With properly managed and maintained ITSs, they have a constant stream of just-in-time requirements to shape their product direction.

ITSs have also enabled *an open and transparent approach to managing SE processes* through issue statuses and public conversations [196]. The community involved in (or just interested in) the development of a particular product can see the workflow of requirements as they are created, discussed, assigned to sprints, worked on, tested, and delivered. This transparency allows users to directly see the version a particular requirement is attached to, and know when and how the outcomes of that requirement will appear in their software. Similarly, users invested in particular Bug Reports can be involved in many ways. They can see if their problem is already reported, or if the similar Bug Reports have similar details. They can also check the status and know if it is being addressed or ignored by the maintainers, and they can also see when the finalised fix for that bug is merged into which version release. This kind of workflow management transparency is a powerful tool for managing an ever-shifting community of users.

Finally, one reason ITSs have gained popularity is the ability to *conduct direct user support* in them, alongside related SE activities. User support differs from other maintenance tasks such as managing bug and security reports in that user support is directly concerned with the thoughts, feelings, and problems the customer is dealing with. In other words, attention is given to this customer and their state of mind, and less attention is given to the software issue they are experiencing. This is a central component of customer relationship management [193]. ITSs support this workflow, and allow direct linking to related Bug Reports, Feature Requests, user stories, and release schedules. This creates forward traceability from user concerns to organisational action, and backwards traceability (rationale) for related organisational decisions.

Overall, the importance of ITSs on modern SE is diverse and impactful across many aspects of the SE lifecycle. Many organisations leverage this power in different ways, and SE researchers continue to strive to understand and improve ITSs as well as associated processes.

2.2.2. Issue Tracking Ecosystems (ITEs)

There are many terms to describe tools such as GitHub [83], Jira [111], and RedMine [191], such as bug tracker [32], issue tracker, Issue Tracking System (ITS), issue repository, and ticket system. Many of these terms exist for historical reasons. For example, "bug tracker" is an older term from a time when these tools stored exclusively Bug Reports. Even recent literature, however, uses terms interchangeably, such as ITS and "backlog" [233]. Seemingly, there is little difference between the underlying concepts these terms describe and how these terms are used.

These terms, despite containing words such as "system", solely refer to the software tool. SE research, however, also involves the study of people, their processes, their plans, desires, and interactions with various tools. ITSs are tools designed specifically to support SE processes such as development and maintenance, they are used by diverse stakeholders, and they are configured

differently across companies, teams, and projects. As such, it is important to consider these factors when researching ITSs, otherwise the results will likely suffer from threats to construct validity, and any recommendations for industry will be missing relevant context factors.

To address these shortcomings, I introduce a new term: Issue Tracking Ecosystem.

> An **Issue Tracking Ecosystem (ITE)** is a combination of an ITS, and all surrounding contextual factors that affect the ITS, including the SE processes used within it, the stakeholders who interact with it, the company they work for, the team they work with, and the project they are working on.

This concept has the key advantage that it explicitly includes the context (compared to *alongside* or *tangential*). This acknowledgement has benefits. The first benefit is simply spreading awareness that these context factors are always involved, and should be considered when studying ITSs. It is often the case that research discussing ITSs ignores potentially relevant context factors, and one reason for this could be the tool-centric names such as "issue tracker" and "bug tracker". By recognising the larger ecosystem encapsulating the ITS, it will be more obvious that researchers should consider these factors in their studies. Second, this explicit modelling acknowledges the interface between ITSs and their surroundings, which draws attention to ITSs in other areas of research. For example, the study of agile methods is largely one of process models and human interaction, while the ITSs play a major role in shaping how agile methods are applied and adopted in industry contexts. Where there is an ITS present and involved, there is an ITE. Finally, by explicitly including the context factors in the definition, studying an ITS *without* those context factors requires an explanation why they are *not* relevant to the study. This encourages a necessary explanation that is often missing from studies of ITE.

Accordingly, I use the term Issue Tracking System (ITS) when referring to the tool itself, and Issue Tracking Ecosystem (ITE) when referring to the tool *and* the surrounding context as described above. I will not use other terms such as issue tracker and bug tracker, as these terms are all synonyms of the more universal term ITS. In general, I will use ITE where possible, and only use ITS when I want to highlight the tool itself.

2.3. Quality, Smells, and Patterns

2.3.1. Quality in Software Engineering

The concept of "quality" in SE research is widespread [176], including requirements quality [60–62, 130, 163, 177, 241], code quality [105], test quality [63], and maintenance quality [27, 250]. A central concept in quality is that of quality attributes [76], which are the specific forms of quality that apply within a given context. For example, quality attributes for requirements include causality [65, 66], conditionals [64], completeness [51], and consistency [163], while code quality includes other attributes such as cohesion, coupling, and fault-proneness [44]. Each

sub-area of quality research features rich and diverse perspectives on what quality attributes exist, why they are important, and how to improve them in automated and manual ways.

ITSs store many forms of SE artefacts, including requirements [98, 233], development items [108, 117], testing tasks [63], and maintenance tasks [27, 250]. The quality of maintenance items in ITSs has received much attention in the past few decades [246], and many notable findings have already been produced [27, 250]. Software quality does not apply to development items in ITSs, since development items are a form of to-do list for the developers. Development items in ITSs are much closer to being requirements than they are to being software. The quality of requirements items in ITSs has received some attention, including Feature Requests [98] and user stories [233]. However, research into the quality of requirements in ITSs is by far the least studied area of ITSs [163].

2.3.2. Smells

The concept of a "smell" in SE nomenclature is that of a potential problem. Fowler and Beck define *code smells* as "violations of coding design principles" [69]. The concept of a smell, however, is broader than just source code, and has been applied to requirements [59], tests [231], and OSS community interactions [201, 220]. A key characteristic of a smell is that it *might* be a problem, and it *might not* be a problem. The act of detecting it does not mean that a problem exists; rather, the situation must be investigated by someone to further understand the existence and extent to which it is a problem. Common code smells include duplicate code, long parameter lists, dead (unused) code, and large classes. While each of these things could be a problem, there are also good reasons for each of these situations to exist in a production code base, depending on the context. Overall, "smell" is a well-known concept that has been applied to several SE concepts.

ITE smells have been proposed by various researchers, including Tamburri et al. [218], Telemaco et al. [222], Qamar et al. [188], Borg [30], Tuna et al. [232], Lüders [137], and Prediger [184]. In accordance with existing literature on smells, they all define and utilise ITE smells as a hint that something *might be* wrong, but requires further investigation to confirm or deny. Before ITEs smells were formalised, ITE *problems* had already been discussed for decades, including work by Halverson et al. [94], Bettenburg et al. [27], Zimmerman et al. [250], and Heck and Zaidman [98]. Given the way in which these articles describe ITE problems, it is clear that they are also referring to smells, rather than problems.

The key advantage to smells is that a minimal amount of knowledge is needed about the context to flag something as a smell. Smells must be further assessed by an intelligent agent (a person or system) to understand if it is a problem in that context. However, this also means that a lot of the important information and work to be done is not captured within the concept of a smell and the real problem of understanding and assessing the existence of an actual problem has not been addressed. For this, a different concept is needed.

2.3.3. Patterns and Antipatterns

Patterns in the SE domain are essential to the condensing, packaging, and communication of knowledge. Design patterns communicate repeatable code solutions to common programming situations [22]. Style guides package and describe agreed-upon stylistic constructs when programming within certain teams or organisations [3]. Architectural patterns guide software architects in the design of larger systems, drawing from knowledge gained from decades of software design [124]. These are just a few examples of SE patterns. The documentation of patterns contains certain elements designed specifically to effectively communicate that knowledge. The common elements of patterns include context, problem, and solution [4]. The *context* positions the mind of the developer to understand when this pattern may be applicable. The *problem* outlines the abstract situation that is undesired, and therefore can be resolved by applying this pattern. The *solution* describes the way in which the problem can be addressed. ITE quality patterns have been proposed by Aranda and Venolia [10] and Eloranta et al. [53]. Similar to that of other SE patterns, these articles outline attributes for each "pattern" that give them structure. The patterns described by Aranda and Venolia [10] are only characterised by a name and a description, and therefore are missing many essential descriptive elements traditionally used within patterns. Eloranta et al., however, provide a list of ITE quality antipatterns, each of which is described with: name, context, solution, consequences, exceptions, and company recommendations [53]. To the best of my knowledge, this is the first time researchers have applied a more formal and complete pattern-like structure to ITE quality.

2.4. Empirical Methods

In this section, I describe the empirical methods applied in this thesis. Each method is explained in detail, touching on all aspects that are relevant for the studies presented in this thesis. In this way, the methods only have to be fully explained once. When presenting specific individual studies I conducted, I briefly summarise the high-level methodological details relevant for the study, but reference this chapter for the majority of the information and reasoning behind the methods themselves. The descriptions of the individual studies, therefore, can focus on the fine-grained methodological details of the specific study being discussed.

I apply four methods for data collection and analysis: interviews, historical data analysis, Thematic Analysis, and ontology building. I only describe two of the four methods in this section, based on their respective complexity and level of notoriety in the SE research community. Interviews and historical data analysis (often called "repository mining") are well-known methods, with highly cited guides on how to apply them in SE research. This is further emphasised by the SE research community's heavy use of methodology articles outlining empirical guidelines for how to plan, execute, and document such studies. Thematic Analysis and ontology-building, however, are less-applied methodological constructs within the SE community that have little

well-known methodological guidance from within the SE community, and therefore result in them being misapplied, over simplified, and often creatively altered without justification [189]. To clarify my use of Thematic Analysis and ontology building, I describe the methodological basis of my studies and which style of these methods I apply in this work.

2.4.1. Thematic Analysis

Thematic Analysis is a qualitative data analysis method for "identifying, analysing, and reporting patterns (themes) within data [...] as it minimally organises and describes data in (rich) detail" [33]. The result of Thematic Analysis is a set of themes that describe the data at a high-level, where each theme is broken down into "codes" that further describe the data. Thematic Analysis is a rigorous data analysis method that enables researchers to draw qualitative insights from textual datasets, both large and small. In this thesis, I utilised both large historical datasets of ITSs and interview notes. I used Thematic Analysis to extract qualitative insights from the data. Large-scale quantitative analysis of datasets can reveal many important generalised findings, but there is often the risk of threats to construct validity if the dataset is not understood well enough. To counteract this threat and gain different perspectives on the data, I applied Thematic Analysis in Chapters 3, 4, 5, and 7.

Thematic Analysis is a process of iteratively reviewing data and forming categories of information until conceptual saturation is achieved. As with all qualitative methods, the goal is not repeatability [189]. The goal of Thematic Analysis is to guide and support the researcher such that the best quality extraction is conducted, *given the reliance on the researcher as a knowledge expert in this task.* The "best quality extraction" is one with high reliability. Thematic Analysis does this in three ways: initial decision guidance when designing the analysis, well-defined phases that iterate over the data in specific ways, and a process of documenting the results that creates traceability from the findings back to individual data points.

Pre-Analysis Decisions

The first step of Thematic Analysis is to explicitly consider four decisions for the analysis: level of detail, style of approach, level of analysis, and epistemological lens [33].

Level of Detail. Consider whether the goal is to achieve a rich overview of the data, or a detailed account of one aspect. For a rich overview, summarising and describing the entire dataset is a priority, and "some depth and complexity is necessarily lost" [33]. The focus is on capturing all meaningful aspects of the data, at the cost of time that can't be spent diving into more specific areas. For a detailed account of one aspect, the analysis is limited to a more in-depth and nuanced investigation of one (or a few) aspects of the entire dataset. The focus is on describing as much (meaningful) detailed as is possible for this subset of the data.

Style of Approach. Consider whether the analysis will follow an inductive or theoretical approach. In an inductive approach, the themes identified are "strongly linked to the data

themselves", which means that the extracted information is data-driven and more representative of a researcher-agnostic extraction process [33]. In a theoretical approach, the analysis is driven by a theoretical or analytical interest, such as an existing model or framework. The analysis is then much more researcher-driven, and tightly linked to this pre-existing theoretical grounding.

Level of Analysis. Consider whether you want semantic or latent themes. Semantic themes represent the surface or explicit meanings present in the data. Latent themes go beyond the surface level meanings, and "identify or examine the underlying ideas, assumptions, and conceptualisations that are theorised as shaping or informing the semantic content of the data" [33].

Epistemological Lens. Consider whether you are approaching the analysis with an essentialist or constructivist lens. Essentialism (and realism) assumes a relatively straightforward relationship between meaning, experience, and language, which means a fairly straightforward extraction process. In Constructionism, "meaning and experience are socially produced [...], rather than inhering within individuals" [33]. This means the analysis is far more focused on the social constructs that could lead to the meaning and motivations within the data.

Phases of Analysis

Thematic Analysis has six phases: 1) familiarising yourself with the data, 2) generating initial codes, 3) searching for themes, 4) reviewing themes, 5) defining and naming themes, and 6) producing the report. These phases are both flexible and iterative. The goal is not to get through them as quick as possible, but rather to understand and utilise the true nature and purpose of each phase until their purpose has been achieved. Understanding the fundamental concept of "saturation" is essential to orchestrating these phases successfully.

Phase 1: Familiarising yourself with the data. It is essential to first read through all the data, at least once—and likely more than that, to gain an initial understanding of what to expect in the analysis. Failing to do this step results in inconsistent analysis results, where the initial work is not as informed as the final work, given the familiarity that you bring to the final work that is not present at the beginning. By fully reviewing all data first, including actively searching for meaning and patterns, you leave this phase with a good understanding of what to expect during your analysis.

Phase 2: Generating initial codes. This phase involves reading through the data, and forming small and specific categories called "codes". These codes are the basic element of the data that can be assessed meaningfully, regarding the studied phenomenon. This phase is complete once all the data has been coded (or actively *not* put into a code). In the case of a detailed account of one specific part of the data, only data relevant to the one specific part of the data needs to be coded, and the rest will be marked as "not relevant".

Phase 3: Searching for themes. This phase involves forming connected ideas across the codes, called "themes". These themes connect the data at a higher, conceptual level, and represent a more analysis-like grouping of the data. Whereas the codes are extracted using

more straightforward mappings of meaning, the themes now involve interpreting the codes, and forming groupings that the researcher believes best fit the analysis goal. This phase is complete once all codes are grouped under a theme (or sub-theme). This phase is more about ideation and coverage than it is about conciseness and a well-formed set of themes.

Phase 4: Reviewing themes. In this phase, the researchers iteratively review and refine the themes for overall consistency and representativeness across the dataset (level one). Then, the evidence under each code is thoroughly reviewed and cleaned such that it is representative of the code and theme itself (level two). This phase is one of the most critical—and yet most overlooked—phases in qualitative analyses (such as content analysis) [33, 43]. While many researchers will naturally iterate over the themes as they create them, it is also important to scrutinise the work *after the forming of the final themes* with a full analysis and cleaning process. Level one of this phase, as described by Braun and Clarke [33], is about reviewing the all codes within each theme, and asking if they form a coherent pattern. While certain codes might have fit best under a certain theme during phase 3, perhaps a comparison of all codes in that theme now reveals that that code is not a good fit for this theme overall. Level one is also about reviewing all themes compared to each other, and evaluating the final group as representative of the overall findings (or not). Level two of phase 4 is an in-depth review of each piece of evidence under each code, with the context of the themes in mind. Every evidence piece is read, and re-considered as a candidate to remaining sorted under this code and theme. The primary reason for this second analysis is the additional final context of the final codes and themes. Evidence that may have made sense under a certain code at the beginning, might no longer make sense with the full context of all themes and codes combined. For example, the code "quick learner" could easily contain the evidence piece "she regularly picks up new knowledge on weekends". However, if the final theme map sorts the code "quick learner" under "machine learning", then the stated evidence piece may no longer fit, since it should also fit with the context of the relative theme. This process is iterative, and is often supported by visual mapping tools such as mind-mapping, facet lists, or taxonomy-like structuring.

Phase 5: Defining and naming themes. This phase is primarily about considering the presentation of the themes: how are they named, how will they be presented, do they tell a cohesive story, do they all fit together well, are they representative of the data and analysis you are now so familiar with? While not the primary purpose of this phase, you can also continue to refine the themes themselves, if the process of considering the presentation leads to an obvious change that should be made. For example, perhaps in the process of naming a theme, you discover that the theme is too broad, and therefore needs to broken down. At the end of this phase you will have a set of presentation themes, each with a name, description, and a well-known set of codes under each.

Phase 6: Producing the report. In this phase, the analysis and findings are described. While writing about data analyses is nothing new to researchers, the importance of this phase for qualitative work should not be underestimated. While quantitative results can be manually

reviewed, re-analysed, and even reproduced, qualitative findings are limited to the interpretation presented by the researcher. With good evidence tracing, qualitative findings can be manually reviewed by other researchers, but the rest relies on the report produced by the researcher. It is critical that the report provides a "concise, coherent, logical, non-repetitive and interesting account of the story the data [tells] within and across the themes" [33]. When discussing the themes, it is important that the "write-up [provides] sufficient evidence of the themes within the data—i.e., enough data extracts to demonstrate the prevalence of the theme" [33]. In addition to the replication package, the researcher should provide "vivid examples [...] which capture the essence of the point" [33]. In other words, the report should describe the findings in such a way that the reader understand the overall findings as the researcher sees them.

2.4.2. Taxonomy and Ontology Development

In this thesis, I develop multiple taxonomies (and a single ontology) using empirical evidence, as well as through secondary studies. While the methodological design, development, and reporting of each taxonomy is self-contained and reported in each respective chapter, there are a few key overarching concepts that need to be explained and clarified. First, I will define my usage of the terms "taxonomy" and "ontology" in this thesis, given the diversity of ways in which the SE research community uses related terms. Second, I will explain the process I followed to construct the taxonomies that were formed from a methodology *other than* Thematic Analysis, and how I constructed the ontology. Third, I will discuss how taxonomies and ontologies relate to theory in SE research.

Terminology: Taxonomy vs Ontology

In 2004, Garshol wrote that "the term taxonomy has been widely used and abused to the point that when something is referred to as a taxonomy it can be just about anything, though usually, it will mean some sort of abstract structure" [81]. This is still the case today, where terms such as "taxonomy", "ontology", "framework", "model", and "topology" are often used interchangeably in SE research [167, 189]. For this reason, I define here the terms "taxonomy" and "ontology" as I use them, within the context of this thesis.

Functionally, the goal of building both taxonomies and ontologies is to *organise information* for use by some *specific actor or system* [91, 92]. The primary means of information organisation for both of them is through a process called *classification*, where objects are grouped relative to their *subjects* [81]. A "subject" is the specific classification chosen for a set of objects. Importantly, the selected subjects must be relevant and useful to the actor or system in some *meaningful* way. For example, if we take a list of sorting algorithms and classify them for a lecturer who will teach them to students, a meaningful classification (the subject) could be average sorting-time complexity (e.g., $O(n)$, $O(nlog(n))$, and $O(n^2)$). Contrastingly, a non-meaningful classification (subject) could be the number of letters in each of their names (e.g.,

4, 5, and 6). These named subjects then become the *controlled vocabulary* [81, 89, 92] through which all future algorithms must be discussed (within the context of this classification).

Taxonomies are best known for their use of *hierarchy* to organise *terms* (objects and subjects). Another lesser known form of taxonomies is that of a *faceted taxonomy*, where terms are organised under *facets* (which can also be objects and subjects). Hierarchies generally form a tree structure (although terms in a taxonomy can have more than one parent [189]), whereas facets form simple single-dimension lists. The relationship that a hierarchical taxonomy uses between terms is called the "broader/narrower" relationship. This is because taxonomies contain *conceptual* hierarchies, where information higher in the hierarchy is "broader", and lower is "narrower" [81]. The fundamental defining concept of a taxonomy is that it provides *structure*, whether that be through hierarchy or facets. As defined by Garshol, taxonomies are "a subject-based classification that arranges the terms in the controlled vocabulary into a hierarchy *without doing anything further*" [81]. The final part of that definition, "without doing anything further", provides a clear and logical boundary between taxonomies and ontologies, since ontologies *can* do other things to support—and go beyond—structuring [89].

Ontologies are best known for their use of properties (also known as dimensions) and relationships to describe terms. An ontology is "a model for describing the world that consists of a set of types, properties, and relationship types" [81]. One possible relationship type is the "broader/narrower" relationship, thus allowing ontologies to contain hierarchical taxonomies—as they often do [81]. The additional tools provided by ontologies, namely "properties" and other "relationship types", allows terms to be further *specified*. For example, we can reuse the sorting algorithms example, and define a subject that is "sorting algorithm", with properties such as "average sorting speed", "fastest sorting speed", and "maximum memory requirement". Thus, ontologies provide more tools for describing terms than taxonomies.

In short, both taxonomies and ontologies *classify*, but taxonomies also *structure* while ontologies can both *structure* and *specify*. Thus, ontologies are a conceptual extension of taxonomies, despite not every ontology containing a taxonomy (hierarchical or faceted). This is useful to keep in mind, since recommendations directed at taxonomies or ontologies therefore apply universally across the two, as long as the concepts they are directed at are understood and applied correctly (those concepts being classification, structure, and specification). In my thesis, I produce an ontology that is based on a central hierarchical taxonomy.

The literature on taxonomies and ontologies uses these words rather interchangeably [81], and I will be citing and summarising the work as they have stated it. However, I will note when I disagree with their use of terminology (based on the foundational definitions above) and therefore apply their recommendations to a different term (but same underlying concept).

Steps to Build and Evaluate Taxonomies and Ontologies

Taxonomies and ontologies are a common research output produced by SE research, but the methods through which they are created lack structure and transparency [125, 167]. In response to this situation, Nickerson et al. [167] (and later updated by Kundisch et al. [125]) outline recommendations for taxonomy construction in the Information Systems domain (which apply rather easily to the constructs in this thesis). While both articles refer to "taxonomy", they are referring to the concept described in this thesis as "ontology" (due to their description and inclusion of properties on the subjects). However, in describing recommendations for ontologies, some recommendations also apply to the construction of taxonomies. Accordingly, I refer to (and apply) their recommendations to my taxonomies and ontology *where it makes sense*.

Nickerson et al. recommend seven primary ontology development phases, which I use to structure the following description of building an ontology [167]. There are two separate paths through the ontology development phases, and I will only describe the path I took: empirical-to-conceptual. Here are the seven phases I applied in this thesis.

Determine Meta-Characteristic. Ontologies are designed to be used for a specific purpose [89, 90, 92, 167]. While they should represent the world as it is, and therefore have an objective description of the objects they seek to describe, there are different ways to frame those objects. This means that there are many different, correct and useful ontologies for describing the same set of objects. However, it is also possible to create unuseful framings, such as the previous example given regarding classification where the number of letters in the name is used to characterise sorting algorithms. Nickerson et al. call this framing of a *useful* concept the "meta-characteristic" of an ontology [167]. It is recommended to pick this meta-characteristic at the beginning of the ontology creation process to avoid "naïve empiricism", a situation in which "a large number of related and unrelated characteristics are examined in the hope that a pattern will emerge" [167]. They also note, however, that the meta-characteristic often "does not become clear until part way through the [ontology] development process" [167]. When picking the meta-characteristic, Nickerson et al. recommend considering who the users are, and how they intend to use the ontology. This lends itself to the functional perspective on ontologies that drives many of the following methodological recommendations [167].

Determine Ending Conditions. The process of creating a conceptual model from empirical evidence is inherently iterative, since any new observation in the data could affect all previous developed concepts [167]. Accordingly, it is necessary to identify ending conditions to this iterative process. In other words, when is the ontology creation process complete? Another important consideration is that of quality. Although the ontology creation process may be finished, objectively, how do we check the quality of the final ontology, and how might we further improve it? Nickerson et al. define two sets of ending conditions: objective conditions and subjective conditions [167]. The objective ending conditions answer the more concrete question of being finished or not, while the subjective ending conditions address the concept of quality.

There are no objective criteria that can outline—with certainty—when a qualitative process such as content analysis is complete, and this also applies to building ontologies from empirical data. There are, however, tips on what things to look out for when trying to "finish" the process, as well as a number of signs that the iteration is likely not done. The recommendations by Nickerson et al. [167] are a mix of those two things. Presented below, verbatim from their article [167], are the recommended *objective ending conditions* for the ontology creation process. When building my ontology, I utilised these objective ending conditions at the end of each iteration, acting as a guide as to when to stop iterating.

- "All objects or a representative sample of objects have been examined."
- "No object was merged with a similar object or split into multiple objects in the last iteration."
- "At least one object is classified under every characteristic of every dimension."
- "No new dimensions or characteristics were added in the last iteration."
- "No new dimensions or characteristics were merged or split in the last iteration."
- "Every dimension is unique and not repeated (i.e., there is no dimension duplication)."
- "Every characteristic is unique within its dimension (i.e., there is no characteristic duplication within a dimension)."
- "Each cell (combination of characteristics) is unique and is not repeated (i.e., there is no cell duplication)."

Due to the context-rich nature of qualitative research—including forming models from empirical data—it is important to use *subjective ending conditions* for iteration guidance. Nickerson et al. propose a set of five subjective ending conditions for building taxonomies. Furthermore, they are clear that these five are the "minimal" ones, and that the researcher "may wish to add more subjective conditions to these based on the researcher's particular view" [167]. Presented here, verbatim from their article [167] (but only showing a portion of the original description), are the recommended subjective ending conditions for the ontology creation process:

Concise "An ontology should contain a limited number of dimensions and a limited number of characteristics in each dimension because an extensive classification scheme with many dimensions and many characteristics may exceed the cognitive load of the researcher and thus be difficult to comprehend and apply."

Robust "A useful ontology should contain enough dimensions and characteristics to clearly differentiate the objects of interest. An ontology with few dimensions and characteristics may not be able to adequately differentiate among objects."

Comprehensive "There are two interpretations of this condition. 1) A useful ontology can classify all known objects within the domain under considerations. 2) A useful ontology includes all dimensions of objects of interest."

Extendible "A useful ontology should allow for inclusion of additional dimensions and new

characteristics within a dimension when new types of objects appear. An ontology that is not extendible may soon become obsolete."

Explanatory "A useful ontology contains dimensions and characteristics that do not describe every possible detail of the objects but, rather, provide useful explanations of the nature of the objects under study or of future objects to help us understand the objects. An ontology that simply describes objects may be of interest initially but will have little value in understanding the objects being classified."

When building my ontology, I refer to these subjective ending conditions list at the end of each iteration, acting as a guide as to when to stop iterating.

Pick the Approach. As with any qualitative conceptual modelling, there are two main approaches: inductive and deductive. Nickerson et al. describe these two approaches as "empirical-to-conceptual" and "conceptual-to-empirical" [167]. Moving forward, I will use the more accepted terms "inductive" (replacing "empirical-to-conceptual") and "deductive" (replacing "conceptual-to-empirical")[2]. It is recommended to use the inductive approach if you have "significant data available" and little knowledge or understanding of the domain, and to use the deductive approach if you have "little data available" and significant understanding of the domain. In the case of having both significant data available and significant understanding of the domain, it is recommended to pick the approach that fits best [167].

Identify Subset of Objects. The researcher needs to describe the population of objects to be studied, and then pick the subset they will analyse as part of the ontology development process. There are many reasons to create an ontology, and therefore there are many potential priorities when picking a population and selecting the subset to be analysed. If the purpose of the ontology is to become theory, then the population should be the actual population of objects that exist under study, and the sample should either be the entire population, or a representative sample. If the purpose of the ontology is to describe some sub-aspect of the population, or act as a tool for particular actors within a system, then the population should be selected accordingly (less need for a representative sample). In such cases, the subset of studied objects "are likely to be the ones with which the researcher is most familiar or that are most easily accessible" [167]. Nickerson describes the potential subset samples as "a random sample, a systematic sample, a convenience sample, or some other type of sample" [167]. In summary, the population and subset need to be selected according to the purpose of the ontology.

Identify Common Characteristics and Group Objects. The researcher then identifies common characteristics across the objects. Importantly, these characteristics should be related to, born from, or perspectives on the meta-characteristic. Without the meta-characteristic as the lens of the analysis, it is too easy to create characteristics that are unhelpful to the purpose

[2]In my opinion, the terms used by Nickerson et al. are misleading because they imply that it is possible to work towards empirical data using conceptual work. The terms "inductive" and "deductive" describe two techniques that produce the same result (some model), but via two approaches (bottom-up vs top-down).

of the ontology, and thus irrelevant to the goal of creating a useful ontology. The characteristics should also separate the objects in meaningful ways [167]. If a characteristic is defined such that all objects have the same value, then the characteristic is not separating the objects. Since the process is iterative, it is okay to create characteristics that are later removed. What might seem to be a good differentiator at first, may have very little discriminatory power over all the objects, and therefore needs to be removed. At the end of this phase, there should be many characteristics that group the objects in different ways.

Group Characteristics into Dimensions to Create the Ontology. In this final phase, the researcher now groups the characteristics into dimensions. It is possible to group these characteristics via statistical methods, but a common approach is to manually group them via visual graphing [167]. These dimensions then become the "properties" of the ontology.

Review Ending Conditions. Ontology development is an iterative process, and as such there needs to both cycles of improvement, and conditions describing when to break this cycle. As described above under "Determine Ending Conditions", Nickerson et al. define a number of objective and subjective ending conditions [167]. Once these conditions have been met, it is then assumed that the ontology is complete and ready to be communicated.

Theory Building in SE Research

Theory enables scientific fields to collect and cumulate knowledge that generalises across many (if not all) known contexts [211]. The advantage of theory is that generalised claims can be formulated, tested, strengthened, and refuted, thus providing a structure for the iterative nature of science itself. While the building blocks of theory are claims, propositions, hypotheses, and empirical evidence (among other constructs) [86, 211], failing to form theory from these building blocks leaves the process stuck in its infancy. Broadly speaking, theory is "a system of ideas for explaining some phenomenon" [86, 189, 211]. There are many views on what theory is [211], as well as different perspectives depending on which epistemological lens is being applied [122, 189]. There are also different conceptualisations and classifications of theory [86, 211]. In this thesis, I am working with the concept of taxonomic theory, the methodological construct most prevalent in my work. I will be focusing on the classification summarised by Paul Ralph in his methodological guidelines for taxonomies in SE research [189]. While Ralph refers to "taxonomy", I interpret the work as applying to ontologies, based on the work's description and inclusion of properties and relationships.

There are three types of theory: variance theories, process theories, and ontologies. Variance theory seeks to explain the world in terms of how one variable (dependent variable) reacts in response to changes in another variable (independent variable). Process theory seeks to explain, understand, and even predict how an entity changes and develops over time [189]. Ontological theory, on the other hand, is a type of theory that seeks to explain and understand entities and systems as they are. Process theories and ontologies are closely related, as three of the four

process theory types *include ontologies* [189]; however, not all ontologies are part of a process theory. While variance theories are the prominent theory type in SE research [189], this thesis focuses on and describes ontological theory.

Ontologies (and taxonomies) are rather common in SE research, but they are often not stated as "theory". There are many potential reasons for this, including the belief that ontologies are not theory [86, 189]. Two additional likely reasons are the lack of understanding as to *why ontologies are theory*, and why some other simple categorisations are not theory. As described above, there are many dimensions and perspectives to theory, such that there is no consensus across scientific communities as to the meaning of theory [189]. However, the broad definition of theory presented above ("a system of ideas for explaining some phenomenon"), requires only that ontologies *explain* some real-world *phenomenon*. Ontologies are commonly built from secondary studies of a phenomenon, as well as from singular empirical studies of a phenomenon (a weaker form of ontology building) [189]. In this way, ontologies are concerned with real-world *phenomena*. While it is possible to create an arbitrary classification of some phenomenon based solely on conjecture or a rigourless process, these are not ontologies [189]. Ontologies are designed to, among other things, "identify, describe and understand the entities and events in a domain" [189], which are all forms of *explaining*. Thus, ontologies are a valid form of theory, when built correctly and described adequately.

The output of SE research is often prescriptive, while theory should be explanatory. These goals should not overlap because they can lead to SE theory including elements of prescription, and can lead to validity issues in the constructed theory [189]. The overlap between Software Development Methodology (SDM)s and process theory is a common place for misconception [189], and relevant for this thesis. When producing a theory that explains how a certain process in SE works, it is tempting to also add how it *should* work if there is an identified deficiency. It is also tempting to take recommendations on how things should be done, grounded in empirical investigations, and frame them as theory. In my thesis, I produce a catalogue of recommendations for SE processes. These recommendations are grounded in investigations of existing SE contexts, including case studies, interviews, observations, and surveys. However, these recommendations are intended to be prescriptive, and therefore are not described as process theory.

Propositions in Software Engineering Research

Propositions are predictions about the world that may be deduced logically from theory [195, 208, 211]. Sjøberg et al. describe the process of theory building in SE as a five-step process, where the first step is to define the constructs of the theory, and the second is to define the propositions of the theory. Propositions are the stepping stone into a fully formed theory, as they represent relationships between the constructs defined in the first step. The combination of the propositions (with some explanations, scope, and empirical evidence) form the singular

construct that is the theory in question [211]. For the purpose of this thesis, however, I would like to focus on propositions themselves.

The first step of theory building is defining the constructs [211]. This is fundamental to both the theory and the propositions, as propositions can only describe relationships between the constructs. One evaluation metric of a good theory is *parsimony*, which is "the extent to which unnecessary constructs and propositions are excluded" [211]. Therefore, it is important to minimise the constructs collected and the propositions constructed. Sjøberg et al. [211] discuss five ways to make a contribution from the perspective of constructs within a theory. For the purpose of this thesis, only the first way applies: "defining new constructs as the basis for building a new theory about some phenomena [...] they might conceive phenomena that have been the focus of prior theories, but in a different way".

The second step of theory building is defining the propositions [211]. These propositions relate the constructs together in different ways. This could be quantitatively, using statistical statements, or more broadly as generalisations to be tested and further refined. For example, a more broad proposition would be to say "the use of a UML-based development method positively affects communication" (example from Sjøberg et al. [211]). A single theory is normally composed of many propositions, but the relative importance of each proposition differs. The importance of the proposition relates to its usage within the theory. Some propositions are more central, and combine with others to form the core of the theory, while others are more tangential, but still part of the whole framing that is the theory overall. I define five propositions in Chapter 6, as a means of beginning the journey to a future theory.

2.5. Summary

ITSs are complex tools that play a fundamental role in SE organisations, including during the requirements, development, and maintenance phases of SE. ITEs are complex systems that involve both an ITS (one or more), and all surrounding contextual factors that affect the ITS. ITEs have quality aspects, and those are related to concepts such as smells and Best Practices. The concept of "quality" for ITE has been framed in different ways, including as patterns, antipatterns, and smells. With the support of different empirical methods such as interviews, historical data analysis, Thematic Analysis, and ontology building, I will investigate the reported difficulties that practitioners have with ITEs in the next chapter.

Part II.

Problem Investigation

Chapter 3.

Practitioner Challenges with Issue Tracking Ecosystems

> Organisation is not a goal in itself, it is a tool. Don't get caught up in the illusion of productivity and get distracted from the actual task.
>
> Unknown

In this chapter, I investigate the claim that ITEs are difficult to interact with by conducting an interview study with practitioners who regularly interface with ITEs. ITEs are commonly reported in pop-culture as difficult to interact with (see Fig. 3.1), but the studied details of such difficulties are lacking. Researching the thoughts and opinions of practitioners is an important part of empirically grounding our research assumptions [71–73, 236]. To understand the difficulties faced by practitioners when using ITEs, I conducted an interview study with 26 industry practitioners. This empirical investigation was conducted in collaboration with my colleagues Dr. Lüders, Christian Rahe, and Prof. Dr. Maalej. I asked practitioners about the problems they face when using ITEs. The results show three key areas where ITS difficulties arise: ITE information problems, ITE workflow problems, and ITE organisational problems. In this chapter, I discuss the details regarding these results, including the contextual complexity of interpreting the results. Overall, this investigation highlights key problems faced by practitioners and draws attention to the context-specific nature of ITEs.

3.1. Research Methodology

My primary objective with this chapter is to understand what problems practitioners face when using ITEs. For this, I have a singular research question:

RQ What *problems* do software practitioners usually encounter when using ITEs?

To investigate this research question, I conducted an interview study with 26 practitioners. The interview involved mostly open questions with the goal of following the interviewee along their thought-processes regarding problems in ITEs. Detailed interview notes were recorded

by two interviewers, which were later transferred to a digital format for analysis. We then analysed the interview notes using content analysis.

Figure 3.1.: Twitter user Chris Bakke jokes about Jira being so complex that an AI—that has already taken over the world—has to ask humans for help.

3.1.1. Interview Participants

Given the objective to gather a broad and rich understanding of ITE problems and contextual factors in practice, we sought a diverse set of participants along five primary dimensions: participant role, years of experience, types of ITSs, company size, and industries. A representative sample was not sought, as we did not aim to generalise the qualitative observations. We contacted SE companies in our network and asked for recommendations for experienced employees who would fit the sampling scheme. All participants worked at SE companies and had at least one year of experience using ITSs. In total, we interviewed 26 practitioners working in Germany, Canada, and Poland, listed in Table 3.1. The participants had a range of work experience from 1.5–25 years (median 7). They held various roles, such as Developer, Manager, and Product Owner. Consistent across them all was the regular task of developing software, albeit less for the participants with more managerial roles.

All participants utilised an ITS in their ITE, and had a range of experience using different ITSs throughout their careers. The majority (14/26) primarily used Jira (in alignment with market share [1, 46, 55]), with six more participants having experience with Jira. Others used GitHub [83], GitLab [84], Trac [229], Azure [14], Bugzilla [38], Trello [230], asana [13], Mantis [148], SpiraTest [214], RedMine [191], BaseCamp [18], Miro [156], and Savanna [200]. The size of their ITSs ranged from just a few hundred up to over a million issues. The company sizes ranged from tens to thousands of employees, covering different industries including automotive, medical, consulting, and energy.

Table 3.1.: Overview of study participants.

#ID	Role	YoE	G	ITS (other known ITSs)	ITS Size	Industry	CS
P01	Product Owner	7	m	Jira	10k-100k	Automotive	L
P02	Developer	4	m	Trac	10k	Engineering	M
P03	Developer	10	m	Trac	10k	Engineering	M
P04	Manager	4	m	Jira	10k-100k	Maritime	M
P05	Manager	6.5	m	Jira (Asana)	100k-1m	Database	L
P06	Developer	6	m	Jira (Trello)	100k-1m	Energy	M
P07	Developer	24	m	Jira (Watson)	1m+	Media	L
P08	Product Owner	5	m	Jira	100k-1m	Automotive	L
P09	Developer	10	m	Jira (Mantis, Watson, Trac)	1m+	Media	L
P10	Manager	5.5	m	Trac	10k	Engineering	M
P11	Developer	8	f	Jira (Custom)	10k-100k	Consulting	M
P12	Manager	11	m	Jira (Brainstorm, Watson)	1m+	Media	L
P13	Developer	3.5	m	Jira (GitHub, Bugzilla)	1k-10k	Engineering	L
P14	Product Owner	6	m	GitLab (GitHub, Jira)	100-1k	Engineering	S
P15	Developer	25	m	RedMine (GitLab)	1k-10k	Research	L
P16	Developer	1.5	f	GitLab (GitHub)	100-1k	Engineering	S
P17	Developer	15	m	GitHub, GitLab (Jira, Savanna)	100-1k	Research	L
P18	Developer	3.5	m	ServiceNow (Jira, SpiraTest)	1k-10k	Technology	L
P19	Developer	16	m	GitLab (RedMine, Jira, GitHub)	10k-100k	Research	L
P20	Developer	9	m	GitLab, GitHub (Jira, BaseCamp)	1k-10k	Software	M
P21	Developer	15	m	Mantis (Jira, Trello, GitHub)	100-1k	Medical	L
P22	Developer	4	m	Jira (Miro, Trello, GitHub)	10-100k	Consulting	L
P23	Developer	23	m	Custom, GitHub	10-100k	Research	L
P24	Manager	15	m	Jira, Azure	10-100k	Consulting	L
P25	Manager	17	m	Jira	1k-10k	Medical	L
P26	Product Owner	3.5	f	Jira, Azure	1k-10k	Engineering	L

Study participants by **Role**, **Y**ears **o**f **E**xperience, **G**ender, currently used **I**ssue **T**racking **S**ystem, **I**ssue **T**racking **S**ystem **Size**, **Industry** and **C**ompany **S**ize [57].

3.1.2. Interview Procedure

We used interviews as the main method for several reasons. First, data analysis and simulation studies are common on ITS data, particularly based on open-source data [58, 202–205]. While useful for characterising recorded phenomena, these studies do not reflect the experiences, priorities, and reasoning of actual SE practitioners. Only focusing on data analysis and lab studies can create a gap between research and practice. Interviews, however, are well-suited to collect perceptions and reasoning of human subjects. In contrast to surveys (which usually include closed questions, focus on representativeness and quantification, and usually assume preliminary hypothesis and sets of variables to measure), interviews usually focus on open questions and enable follow-up clarifications. This is particularly important for us to gather the *context* when a certain problem matters. Interviews also enable asking *"why" follow-ups*, which is crucial for understanding practitioners' perceptions.

When designing the interviews, we aimed at maximising realism, diversity, and reliability. Realism implies that our observations should reflect real ITE interactions in real project envir-

onments. We thus decided to gather insights only from practitioners who worked for years in industry, and excluding students and researchers. By maximising diversity, we tried to cover diverse perspectives, including diverse roles, companies, industries, and ITSs. The aim was not to seek for representative results, rather, we aimed to draw a comprehensive picture of ITE problems. To maximise the reliability of results, we took measures to reduce researchers' interference and potential bias as far as possible—as the goal was to summarise the perceptions of practitioners, which might or might not affirm current findings in research.

We conducted 1-hour semi-structured interviews over video calls, guided by an interview protocol. To encourage an open dialogue regarding their true thoughts on the quality of their company's ITE practices, we refrained from recording the interviews. To minimise observer bias, *two interviewers* were present in each interview session, who both could ask follow-up or clarification questions. We used slides to guide the interview sections. The two interviewers met within ~24 hours of the interview for 15–60 minutes to discuss the results and align the notes. 15 interviews were conducted in English and 11 in German. The language was decided based on the preference of the participant. All German interviews were conducted by native German speakers who are also fluent in English, who later translated their notes into English.

Each interview consisted of three main parts: problems related to using ITEs, perceptions of smells related to ITE Best Practices, and opinions on tooling to manage these problems and smells. Only the first of these three parts is discussed in this chapter, whereas part 2 is discussed in Chapter 7 and part 3 is discussed in Chapter 9. We started with a short welcome session (~10 minutes) where we introduced ourselves and the study and asked about their role, experience, and work context. We only introduced key concepts when they were necessary to avoid biasing their answers to earlier questions. Relevant to this chapter, we asked the interviewees about challenges they encountered during their work with ITEs.

3.1.3. Analysis

Following the interviews, we digitised the 3,675 recorded meeting "notes" for analysis.[1] We gave each note a unique ID based on the participant, interviewer, and interview question, allowing for evidence tracing to support the analyses. We conducted a Thematic Analysis of the entire set of notes to produce the qualitative findings for the research question.

Thematic Analysis of Interview Notes. The findings of the interviews were extracted using Thematic Analysis [34].[2] We independently read through all the digitised notes, to familiarise ourselves with the data (phase 1). Then, we independently extracted preliminary codes (phase 2). We then met to compare, discuss, and group the codes into a single agreed-upon set (phase 3). During these shared meetings, we also began phase 4, where we created an initial set of themes based on our shared understanding of the codes. All four researchers met

[1] Each "note" corresponds to a complete thought, usually a sentence.
[2] Dr. Lüders, Christian Rahe, and I conducted phases 1–6, while Prof. Dr. Maalej was involved in phases 3–6.

to discuss and refine the themes and codes based on our shared knowledge. We then named and defined the themes (phase 5), and selected representative examples (phase 6).

Replication Package. I created a replication package as part of this thesis [160]. This package includes all research artefacts discussed in this chapter, except for the digitised meeting notes. Statements and knowledge about ITEs can be quite critical (both for the business and the individuals), and so to protect the privacy of the interviewees, the digitised notes are not archived. I do, however, share the remaining artefacts which have the IDs embedded in them, allowing for interpretation of the observations in context: you can trace analysis to the participant table and connect to their anonymous context factors.

3.2. Results: ITE Common Problems

In this section, I describe the recurrent problems the participants observed in their daily work with ITEs. These results are from the Thematic Analysis applied to the participant statements.

3.2.1. ITE Information Problems

Participants described difficulties retrieving information from the ITE and maintaining a general understanding of the overall ITE state.

P1. Missing Issue Information. Eighteen participants mentioned problems related to the accuracy and completeness of information in their ITSs. This includes information that is incorrect, irrelevant, or missing, as well as vague definitions and insufficient details in issue descriptions (confirming results by Zimmerman et al. [250]). These problems were more often mentioned by developers (12/16) than managers (2/6). P07 said that the "main problem is the people who did not enter enough information". There is simply too much to be filled in across too many diverse and irrelevant workflows, so users revert to skipping the field or filling in incorrect information. Without the correct information, resolving issues takes a longer time due to required clarifications, or may not happen at all. One context factor for this appears to be when ITEs are used by many untrained or external people.

P2. Issue Overload. Seventeen participants mentioned the problem of too many issues (both open and closed) within their ITSs. P18 described "[getting] lost navigating around Jira". The consequences of this problem include a lack of awareness of existing issues, leading to duplicates and overlapping issues, missed links, and issues being forgotten. As context factors, participants stated older and larger ITSs (as they have likely accumulated more issues). Issue overload appears to affect managers more than developers.

P3. Zombie Issues. Twelve participants mentioned the problem of inactive and abandoned issues within their ITSs, which some referred to as "Zombie Issues". These participants (primarily developers), described the problem as a lack of attention towards issues that were still open, but were waiting for some action to be taken. Specific causes for this problem include when the

reporter and assignee are the same person (accountable only to themselves), and low-priority issues (minimal pressure to act). P07 said some issues get ignored, and that "it happens to all developers" due to a "focus on other topics". Zombie Issues may lead to a clogged ITS, and duplicate issues (due to unawareness of existing issues). The context factors for this problem include older and larger ITSs (where the stressor is more issues), as well as multiple disjoint teams working loosely together (which leads to uncertainty about respective responsibilities).

P4. Ineffective Search. Ten participants mentioned the problem of ineffective search within their ITSs. They described difficulties searching for specific issues, as well as classes of issues such as open Epics associated with a certain project, due to limitations of the search features in the ITS. Even with sophisticated filtering, the exact phrases required to locate the issues were not easily obtainable. This would result in the search returning no issues, or too many issues related to non-important aspects of the search term (confirming findings from Heck and Zaidman [98]). P09 mentioned that "there are many synonyms, which makes it harder to find something". Difficulties in locating relevant existing issues lead to a less connected ITS, as well as duplicate issues. Ineffective Search is particularly severe in larger ITSs.

P5. Lack of Comprehensive Overview.

Ten participants mentioned the lack of a comprehensive overview. They described difficulties gaining a holistic understanding of their ITSs, as well as their own tasks and responsibilities. They expressed a desire for a better grasp of the big picture and the dependencies between issues. P08 stated that "it is hard to keep an overview of the huge backlog, and keep the projects separate from each other". Participants described the cause as missing features in their ITS. While some ITSs facilitate custom dashboards, these must be created by the users, who are themselves often not sure what to visualise. The consequence of a lack of an overview is the repeating need to manually find things, as well as the need to remember (or write down) exactly what to keep track of. The desire for such a dashboard was mentioned by half of the managers (3/6) and product owners (2/4), but less than a third of developers (5/16).

3.2.2. ITE Workflow Problems

Most participants described a complex and nuanced trade-off between the bloat that exists for more complicated workflows, and the need for more complete workflow coverage. This includes the problems: Workflow Bloat, Missing Issue Information, Information Islands, Divergent Needs, Lack of a Fitting Workflow, and Unclear Workflow.

P6. Workflow Bloat. Nineteen participants mentioned the problem of complex workflows within their ITE, which some referred to as "Workflow Bloat". This includes too many issue fields, required steps, and over-engineered workflows with numerous edge cases and excessive maintenance overhead. P13 mentioned that there are "too many link types, a whole drop-down menu". Another challenge is the amount of information that must be entered while creating an issue, which can be tedious and time-consuming. P17 reported a "fine granularity

of issue fields with a very granular workflow", which sometimes leads to "fitting yourself into the workflow, instead of just doing the work". This can be particularly problematic when the required information is not known at issue creation time, or when the ITS is configured with overly strict rules. P02 and P20 mentioned examples where one small bug fix needed several hours of documentation. The consequences of Workflow Bloat include confusion regarding what needs to be done and uncertainty regarding the importance of the required information. P18 mentioned that in Jira "there is so much information presented that is often not used". This can lead to users ignoring important fields, or entering the wrong information. This can also result in fields having no meaning, e.g., one participant mentioned that too many incoming Bug Reports were marked as the highest priority to get attention [197], but are actually not a high priority. One context factor for Workflow Bloat is older ITSs, given the possibility for outdated fields and processes. Another is when many teams with different internal workflows all use the same ITS configuration. A more powerful ITS can also be a context factor, given the possibility for more complex workflows. As P05 mentioned, "Jira tries to mimic a perfect world where everything is known".

P7. Lack of Workflow Enforcement. Thirteen participants mentioned the problem of a Lack of Workflow Enforcement in their ITSs. This problem includes a lack of enforcement of required fields, accepted norms of ITS usage, and workflow steps that are assumed to be consistent. About half of the participants (more managers than developers) mentioned that ITSs can allow "too much freedom", resulting in erroneous user input, such as misusing specific properties. P06 mentioned that the "environment property is a large text field where sometimes the description is erroneously entered". Consequences of a Lack of Workflow Enforcement include missing or incorrectly located information, and skipping statuses, e.g. "testing" or "quality assurance" of implementations. However, care must be taken when addressing Lack of Workflow Enforcement through automated restrictions, as they may lead to additional Workflow Bloat, especially in the presence of an Unclear Workflow. Moreover, if there are exceptions to the enforced rule, or the rule affects teams with Divergent Needs, a Lack of a Fitting Workflow may arise from preventing certain behaviour.

P8. Lack of Workflow Support. Six participants mentioned the problem of a Lack of Workflow Support in their ITS. Examples of desired automation include setting an issue from "in progress" to "in review" once a fixing commit was made, or closing a task once all sub-tasks were closed. The consequence of this lack of automation is more manual work needed, which may then be done incorrectly or not at all. If automation support is implemented, workflow automation can help ease the problem of Workflow Bloat by taking away manual steps and therefore making the process less tedious.

P9. Lack of a Fitting Workflow. Five participants mentioned the problem of a Lack of a Fitting Workflow in their ITSs. They described limitations in issue fields and workflow options. P26 said, "there are so many exceptions in real life; the process is valid for 90% of use cases, but the 10% of cases do not fit, so you have to cheat the process to make it

work for these 10%". A consequence of this problem is information being stored in the wrong location or lost altogether, as there is no appropriate place for it. This lack of information can lead to uncertainty. Lightweight ITSs might not offer advanced customisation options and could potentially be more at risk of experiencing this problem. Working to address a Lack of a Fitting Workflow has the potential to incur Workflow Bloat through newly added fields or options. Conversely, Workflow Bloat itself may lead to Lack of a Fitting Workflow, if strictly defined workflows make it difficult to represent edge cases in the ITS. Users need to find a balance between Workflow Bloat and Lack of a Fitting Workflow.

P10. Unclear Workflow. Five participants mentioned the problem of Unclear Workflow in their ITS. This includes unclear definitions, vague meanings, and differing definitions of issue fields, e.g., what do "done" and "fixed" mean for a bug versus an epic? P05 mentioned, "Jira tries to establish a naming standard (e.g., blocked and resolved), but what does "resolved" actually mean?". This problem may lead to users spending more time than intended on the issue documentation, or alternatively skipping steps because of decision fatigue.

3.2.3. ITE Organisational Problems

Some participants described problems involving the collaboration across organisational contexts and individual personnel.

P11. Lack of Mindset & Discipline. Fourteen participants mentioned the problem of a Lack of Mindset & Discipline within their ITS. These participants explained that users must adopt the right mindset and discipline to keep the ITSs up to date, requiring motivation and integration into daily work. P15 and P22 both mentioned that good discipline is needed. P22 elaborated, saying that "ITSs must be used properly to be successful, this has nothing to do with the ITS itself". This problem also includes inappropriate communication, waiting for people, or difficulty choosing an estimate or due date. Managers are often responsible for organising issues and keeping them up-to-date, which can lead to developers creating issues with lower quality. It is problematic if other users do not see the value in correctly creating and updating issues if the processes are too complex or tedious, or if the overhead is too much. As Meyer et al. [155] observed, developers feel productive when they close many or big tasks, whereas creating or organising issues feels unproductive. If the ITS is not kept up to date, this problem can lead to other issues, such as Zombie Issues or Missing Issue Information.

P12. Scoping Issues is Hard. Eleven participants mentioned that Scoping Issues is Hard in their ITSs. They described difficulties breaking apart and appropriately scoping large issues. The participants described that ITS users may struggle with uncertainty and ambiguity, resulting in issues being scoped incorrectly or lacking clarity in their scope. Managers were affected by this more than developers. Users might be uncertain when an issue is done and close issues too early, leading to issues needing to be reopened. As P14 pointed out: "what is the definition of 'done' for this work?".

P13. Divergent Tracking Needs. Ten participants mentioned the problem of divergent needs in their ITE, particularly across teams and projects. This includes variances in issue fields and available properties, but also entire workflows supported by the ITS. As described by P17, there is a "heterogeneous usage of Jira, even within the same project, which means no one is following the same workflow". The consequence of Divergent Needs is that ITSs become too diverse with the number of available options and start exhibiting Workflow Bloat, thereby creating confusion as to the correct options for a given workflow. This can lead to communication issues and a lack of understanding about the processes being used by different teams, including their own. One context factor for this problem is larger companies with many projects and teams, all using the same ITS instance.

P14. Information Islands. Ten participants mentioned the problem of multiple disjoint systems and missing integration in their ITE, leading users to duplicate, split, or omit information across systems. Such disjoint systems include Git, communication and documentation tools, and even multiple ITSs being used in parallel. A common reported problem was the use of multiple ITSs within the same organisation, with slightly different use cases (e.g. one public and one private ITS), which lead to many duplicate and missing issues across the systems. As P20 mentioned, there are "multiple sources of truth, including Slack and Jira". P19 similarly mentioned, "Many things [are] not stored in the ticket: inside other communication tools, such as MatterMost". One cause of this problem is the use of many similar systems, without useful integration. The consequences of Information Islands include duplicate and missing issues, as well as increased effort required to create, maintain and retrieve the information across systems.

P15. Difficult Customisation. Eleven participants mentioned the problem of customising an ITS to the workflow and needs for specific projects and teams. Some ITSs, such as Jira, can be too complex and have too many features, leading to "technology overload" [242] and confusion about how to use certain features correctly. P17 said, "being able to configure Jira properly is a difficult task that requires advanced knowledge of development workflows and software development practices". P21 said that it is "difficult to use these tools out-of-the-box" as it "requires a lot of customisation". The more familiar a Jira administrator or user is with the features, the fewer problems are likely to occur.

P16. Issue Linking is Cumbersome. This includes issues where linking requires multiple clicks or where the link type cannot be easily changed. This was only mentioned by a few participants. They confirmed that creating links in some ITSs, such as Jira, can be cumbersome, as it is only possible to add and delete links, not change their type or the involved issues. P12 said, "Link types can't be changed. If I set a link type wrongly, I need to set a second link [...]". Incorrect and missing links make it harder to gain a holistic overview (Lack of Comprehensive Overview) and can hinder efficient navigation and organisation within the ITSs.

3.3. Related Work

Quality of Issue Reports. The quality of issue data (P1) is a critical prerequisite for effective issue resolution (P8–P10). Bettenburg et al. [27] and Zimmerman et al. [250] first studied gaps between information contained in Bug Reports and information needed by developers to fix the bugs (P1, P7, P8, P10, P11, P13). Since then, plenty of follow-up works investigated particular aspects of Bug Report quality. Huo et al. [104] compared Bug Reports written by developers vs users and found a significant difference (P10, P11) that impacts prediction models. Chaparro et al. [42] focused on the quality of steps-to-reproduce in the reports, while Davies and Roper [47] focused on observed behaviour and expected results (P10, P11). These works led to improvements in ITSs for creating good Bug Reports: such as assisting reporters to include relevant and essential information (P1, P8, P10), incentivising good quality Bug Reports (P8), and merging additional information into existing issues [36, 118]. My work is complementary. Instead of focusing on the reporter perspective, I studied the ITS process as it interferes with the work of developers, managers, and product owners. I also studied various issue types beyond Bug Reports, such as requirements issues and development tasks.

Only a few studies investigated the quality of issues beyond Bug Reports. Heck and Zaidman [100] defined three levels of Feature Request completeness: basic, required, and optional. They found that all Feature Requests fulfilled the basic completeness, but only 54% fulfilled "required". Seiler and Paech [207] ran interviews on the problems with Feature Requests in ITSs and found that unclear feature descriptions, insufficient traceability, and fragmentation of feature knowledge are common in practice. My results confirm their findings and add more context (e.g., when details or links matter in practice and why). Moreover, I cover other issue types such as user stories, epics, and tasks—which are frequently used in modern ITSs [164].

Mining Issue Repositories. Recent ITS datasets include millions of issues with thousands of comments and links [164]. Several studies have highlighted that large data volumes in ITSs can be overwhelming for manual handling [8, 20, 79, 192] (P2, P4, P5). Thus, routine tasks such as prioritisation and planning become challenging and time-consuming [21, 56, 98] (P8–P10).

There are numerous ITS mining studies, often to retrieve relevant issues or predict particular fields. Common research goals include issue classification, issue assignment, issue prioritisation, or duplicate detection [41, 251] (P4). Classification aims to identify or correct the issue type (bug, Feature Request, or enhancement) based on issue properties using Natural Language Processing (NLP) and machine learning [152, 178]. Issue assignment aims at predicting an assignee for an issue, e.g., by analysing the change history, affected components, code ownership [244], issue text and code similarity [215], or previous contributions [194]. Issue prioritisation aims at predicting the severity, priority, or ranking of issues, based on issue properties [107, 127, 134, 151, 226, 227]. I used a different research method and focus on a different perspective. Instead of data mining, I conducted an interview study for an-depth understanding of how practitioners use issues and issue properties, as well as problems they encountered.

3.4. Discussion

Context is key. Our results highlight that there is no "one size fits all approach" for intelligent information retrieval and automation for ITEs. It all depends. Which information is expected in certain issues and which not? Which automations are considered to hinder stakeholder's work and which not? What assistance is considered helpful? How should the issue and its attributes best evolve? Should certain workflows be strictly enforced? Should certain discussions and collaboration take place in certain tools? The answers to these questions depend on the organisational context, much of which is embedded directly in the ITE. What may sound like a simple finding is particularly crucial for future SE research. Automated approaches, e.g. to retrieve related or duplicate issues, predict the priority, or assign the issues, have been so far rather universal. Our results suggest that researchers should consider these confounding context factors more carefully, as they might strongly impact the evaluation of intelligent ITS solutions. Our results include some context factors, such as the issue type, the role of the stakeholder, the age and size of ITSs, and the practices followed by a project or a team. How universal the impact of context factors on certain ITEs or types of projects remains unclear. I hope that our work inspires follow-up studies to formalise, quantify, and measure the impact of these context factors. The main takeaway is that context is key, and needs to be considered when confronting these problems and attempting to improve the quality of ITEs.

Conflicting perspectives. The participants shared conflicting perspectives on many of their problems with ITEs, which comes down to the context-sensitive nature of these problems. The clearest example is the difference between the problems Workflow Bloat and Lack of a Fitting Workflow: the more workflow support that is added for one set of stakeholders, the more workflow bloat that is added for a different set of stakeholders. It is tempting to imagine an ideal world where a tool supports all relevant workflows, but does not have anything extra that would constitute workflow bloat. However, this assumes that such a tool could exist, that the correct configuration is known, and that there are no conflicting configuration settings. From the interview results, it is our understanding that the problem is not about the tool itself, but the context of the stakeholders using the tool. Multiple diverse groups of stakeholders (within the same organisation) all trying to use the same ITS instance will ultimately find themselves lacking some workflow process, and also experiencing workflow bloat. One such solution to this could be to use multiple ITS instances, but this can create siloed workflows where cross-cutting operations like search may not function correctly. Another solution may lie in automation support, as hinted by some interviewees.

Data heterogeneity. The heterogeneity of ITSs and issue data in practice also sheds light on evaluating machine learning models trained on issue data. This heterogeneity influences the issue data, its structure, and semantics. It seems thus crucial that models (e.g., to predict duplicate issues or who should fix a defect) should be evaluated across different ITEs and different contexts—not only on popular benchmarking datasets such as GitHub or Bugzilla.

In this line, Lüders et al. [138] recently showed that duplicate prediction models developed on Bugzilla data are much less accurate with Jira data: in this case due to the greater variety of issue links in Jira. I argue that researchers should either precisely scope the ITE context targeted by their ML models or extend the evaluation to heterogeneous ITS contexts.

Implications for practitioners. Our results also have implications for *SE practitioners* concerning 1) the archiving of issue data, 2) the configuration of the trackers, and 3) the training of their users. First, managers and administrators should carefully consider limiting the size and diversity of issue data, particularly in large long-lasting projects. Archiving old issue data could reduce the overload and stress when searching for relevant issue data. Second, configuring ITSs seems like a highly crucial task that should be revisited regularly, possibly consulting different stakeholders. If, for instance, old issue types or link types are not used any more, they should be cleaned from the trackers. Our results suggest that restricting users too much can make them frustrated, but if the degree of freedom is too high with little validation, many mistakes can occur. Too many issue properties and value options can also become counterproductive. For example, Herraiz et al. [101] argued for simplifying the issue report form in Eclipse. The degree of detail and freedom depends on the specific needs and processes of the team or organisation. Moreover, most ITSs offer powerful workflow configuration features, for instance Jira for automation.[3] Our results suggest that some teams seem unaware of (or unwilling to) use these features. Third, training and educating software practitioners about ITS processes seems crucial. Overall, the use of ITSs seems ad-hoc without common principles, rules, or (explicit) guidelines. This can be tackled in part by generating awareness and regular training, for example, on how to create and evolve issues, what are good and bad practices, and what are powerful options to configure and analyse ITSs.

3.5. Summary

I identified common ITE challenges by interviewing 26 experienced practitioners from diverse domains. Identified challenges include the efficient retrieval of relevant information from the ITE, maintaining and navigating workflows within the ITE, and managing the ITE usage on an organisational level. These challenges become increasingly severe as ITSs age and various stakeholders with various habits, roles, and practices get involved. Default search features and universal issue property recommenders only address these challenges in part. These results motivate researching intelligent ITE approaches based on mining issue data and considering *context factors* such as issue types, practitioners roles, previous interaction with the issues, and diversity of the ITE. The results also highlight how software teams are striking the balance between automation and flexibility in ITEs. While some problems are directly related to the tooling, many problems are more related to the *context-specific* way in which people work. The

[3] `https://www.atlassian.com/software/jira/guides/expand-jira/automation-use-cases`

results highlight some needed areas of improvement in ITEs, which may come in the form of improved tooling, but will more likely come in the form of automation support and well-defined processes. These results lead to questions regarding the complexity and diversity of ITEs. What kinds of information are stored inside these systems? How often do the issues evolve? I investigate and answer these questions in the following chapter. These findings also raise questions about how to address these problems. Automation seems like a necessary next step, but what recommendations are appropriate, and under which circumstances? I seek to address this issue in Part III, beginning in Chapter 6.

Chapter 4.

Artefacts and Activities within Issue Tracking Ecosystems

> The secret of getting started is breaking your complex, overwhelming tasks into small manageable tasks, and then starting on the first one.
>
> Mark Twain

In this chapter, I investigate the types and prevalence of different artefacts and activities in ITSs. ITSs have existed for many decades, and began as a tool to track bugs in software as "Bug Reports" [27]. As SE shifted towards Agile practices, involving less documentation and more iterative improvement, there was a need for more lightweight documentation and even better support for bug fixing. The ITS, a tool commonplace in most SE organisations, slowly grew to accommodate those needs. Artefacts like Epics, User Stories, and Support Tickets are now represented as "issues" within ITSs. However, the full extent of artefacts and activities represented in ITSs is not well known. In this chapter, I first compile a dataset of 16 publicly available Jira ITS repositories. The dataset has 352 unique issue types across the 16 ITSs. I conducted a Thematic Analysis to investigate, understand, and categorise the available issue "types" across the organisations' ITSs. From this analysis, I found 12 artefact types (thematic codes), which I categorised into 3 software activities (thematic themes). In this chapter, I produced a thematic mapping of the artefacts and activities represented in 16 different Jira ITSs. I also collected and publicly released a dataset of 16 public Jira ITS repositories, available on Zenodo.[1] The results reveal a diverse ecosystem of artefacts and activities within ITSs. The next chapter investigates the information within issues, as well as the evolution frequency, time, and ownership of different artefacts and activities in ITSs.

Publications. This chapter contains portions of my collaborative publication, some copied verbatim [164].

[1] `https://doi.org/10.5281/zenodo.5882881`

4.1. Research Methodology

My primary objective in this chapter is to explore and understand the different artefacts and activities documented within ITSs. I define three research questions that guide this work.

RQ1 What artefacts and activities are documented within ITSs?

RQ2 How prevalent are the artefacts and activities within ITSs?

RQ3 Which artefacts and activities co-occur within ITSs?

To investigate the artefacts and activities within ITSs, I first collected and curated a dataset of 16 public Jira repositories. This dataset serves as the foundation for much of the research presented in this thesis. I discuss the details of the collection and curation of this data in the next section (Section 4.2). Jira is a tool composed of "issues", where each issue is of a certain "type". This type can be, for example, "Bug" or "Feature Request" [103], to name a few popular and well-studied types in SE research. The issue type indicates the general organisational category the issue belongs to and the workflow processes it must undergo from "open" to "close", thereby clearly separating its use from issues of other types. By investigating the utilised issue types in a given set of ITSs, I can uncover the artefacts and activities contained within the ITSs.

The difficulty of this task, however, is that Jira is a highly customisable tool, where each organisation defines which of the default options they will use, and which things they will customise. Therefore, issue type usage is not standardised, despite the likelihood of similar underlying organisational processes. A preliminary analysis of the dataset (presented below) revealed 352 unique issue types across the ITSs (see Table 4.1). Given this large number of unique issue types, it is unclear which issue types are used by each ITS, how to group them, and then how to unify them across the ITSs for more general analyses.

To address this investigation and unification problem, I conducted a Thematic Analysis [33, 43] of all issue types across all 16 of the ITSs to provide a unified mapping between the issue types. I followed the recommendations by Braun and Clarke [33] and Cruzes and Dyba [43]. Before beginning the analysis, I first decided on three major factors of the analysis: the type of analysis, the style of approach, and the level of analysis [33]. For the type of analysis, I chose a Rich Overview. The goal of this analysis was to get an overall understanding of the artefacts and activities contained within ITSs, which requires all facets of the data to be considered. For the style of approach, I selected a Theoretical approach. I was interested in mapping the ITS data as it related to contemporary SE artefacts and activities (theoretical), rather than finding emerging ideas that might be imbedded in the data (inductive). Despite the choice for a theoretical approach, reviewing and understanding the data was still an integral part of the analysis, just with the top-down theoretical lens of existing SE constructs. For the level of analysis, I selected Latent Themes. I was interested in understanding more than just what the issue type name, summary, and description had to say, I was interested in why these issue types

existed, and what they were being used for. This type of analysis, which requires me to interpret beyond the presented semantic information, required a deeper layer of interpretation known as Latent analysis. The final choice, then, was a Theoretical Thematic Analysis providing a Rich Overview that produced Latent Themes.

The primary analysis was on the names of the issue types. The issue Summaries and Descriptions were also analysed to cross-reference the expected artefact type based on the name of the issue. I followed a five phase Thematic Analysis process, as recommended by Braun and Clarke [33] and Cruzes and Dyba [43]. First, I familiarised myself with the data by looking at the issue type descriptions in the repos and 50 example issues for each type. Second, I merged issue types with similar names and meaning, and identified initial phrases or codes describing the types. Third, I iterated through the codes and example issues and identified an initial set of themes. Fourth, I reviewed and reduced the themes during two peer-discussion sessions with researchers familiar with ITSs. I finalised the themes by forming definitions for each.

4.2. Issue Tracking Dataset

Here I describe the dataset I collected for this research, which later served as the foundation for many of the empirical investigations I performed.

4.2.1. Why Jira

Over the last two decades, SE research has intensively studied issues and ITSs [16, 96], often based on Bugzilla [38, 129] and GitHub [83]. The primary research focus has been on specific issue types, such as *Bug Reports*, to either improve the quality therein [27, 250] or to predict properties such as severity [126, 127], assignee [110], and duplicate reports [50, 97, 238]. Another prominent but rather understudied ITS is Jira. Jira [111] is an agile planning platform that offers features such as scrum boards, kanban boards, and roadmap management. Jira's ticket-centric design mimics that of GitHub[2], GitLab[3], and Bugzilla. The benefits of Jira over other ITSs include a history of issue changes, complex issue linking networks, and a diverse set of custom field configurations across organisations. According to 6Sense [1], Datanyze [46], and Enlyft [55], Jira is by far the most popular tool in the ITS and agile project management markets.[4] Yet, it is under-represented in SE research: Google Scholar *article title search* (1980–2023) for "GitHub" returns 5,840 results, "Jira" returns 747.[5] I believe the lack of wide-spread research on Jira is due to the lack of available Jira data to study. While Ortu et al. released a dataset of four Jira repos in 2015 [173], our dataset is larger and more diverse.

[2] `https://docs.github.com/en/issues`
[3] `https://docs.gitlab.com/ee/user/project/issues/`
[4] To the best of my knowledge, there is no empirical evidence on the magnitude of Jira usage in industry.
[5] GitHub: `https://scholar.google.com/scholar?q=allintitle%3A+Jira&hl=en&as_sdt=0%2C5&as_ylo=1980&as_yhi=2023` & Jira: `https://scholar.google.com/scholar?hl=en&as_sdt=0%2C5&as_ylo=1980&as_yhi=2023&q=allintitle%3A+GitHub&btnG=`

Overall, Jira is a more popular and complex ITS, which opens up many important avenues for research. Given the feature-rich nature of Jira, other ITSs tend to offer a subset of the features available in Jira. Findings that apply to Jira datasets, can likely be applied to other ITSs. Additionally, Jira keeps track of historical changes to issues, which is a useful tool for researchers. This enables a special type of evolutionary investigation that other ITSs either do not offer, or only offer in partial ways. For these reasons, I chose Jira as my primary data source and tool for investigating ITSs.

4.2.2. The Jira Dataset

Table 4.1.: Dataset consisting of 16 public Jira repositories.
Column names: **D**ocumented **I**ssue **T**ypes; **U**sed **I**ssue **T**ypes; **D**ocumented **L**ink **T**ypes; **U**sed **L**ink **T**ypes; **Ch**anges per **I**ssue; **Co**mments per **I**ssue; **U**nique **P**rojects

Jira repo	Born	Issues	DIT	UIT	Links	DLT	ULT	Ch/I	Co/I	UP
Apache	2000	1,014,926	48	49	264,108	20	20	10	5	657
Hyperledger	2016	28,146	9	9	16,846	6	6	12	2	36
IntelDAOS	2016	9,474	4	11	2,667	12	12	15	3	7
JFrog	2006	15,535	30	22	3,303	17	10	9	1	35
Jira	2002	274,545	52	38	110,507	19	18	21	3	123
JiraEcosystem	2004	41,866	122	40	12,439	19	18	15	2	153
MariaDB	2009	31,229	11	10	14,950	-	6	12	-	24
Mindville	2015	2,134	2	2	46	-	4	3	-	10
Mojang	2012	420,819	1	3	215,821	6	5	8	2	16
MongoDB	2009	137,172	31	37	92,368	15	13	17	3	95
Qt	2005	148,579	19	14	41,426	10	10	12	3	38
RedHat	2001	353,000	74	55	163,085	23	19	13	2	472
Sakai	2004	50,550	43	15	20,292	7	7	10	4	55
SecondLife	2007	1,867	10	12	674	9	5	30	8	3
Sonatype	2008	87,284	16	21	4,975	10	9	7	4	17
Spring	2003	69,156	13	14	14,716	7	7	8	3	81
Sum		2,686,282	485	352	978,223	180	169			1,822
Median		59,853	18	14	15,898	10	10	12	3	37
Std Dev		251,973	31	16	81,744	6	5	6	2	178

I collected data from 16 public Jira repositories (repos) containing 1,822 projects and 2.7 million issues. The dataset includes historical records of 32 million changes, 9 million comments, as well as 1 million issue links that connect the issues in multiple ways. Table 4.1 shows an overview of the data. The table lists the following attributes for each Jira: the year it first was used, the number of issues, number of documented and used issue types, number of documented and used link types, number of changes per issue, number of comments per issue, and number of unique projects.[6] Apache is the largest repo with 1 million issues and 10.5 million changes. Mindville and SecondLife are the smallest Jira Repositories (Jira Repos) with ~2,000 issues

[6]Documented issue types may be removed over time, but their use is still recorded in the data, leading to a lower number of documented issue types than used issue types.

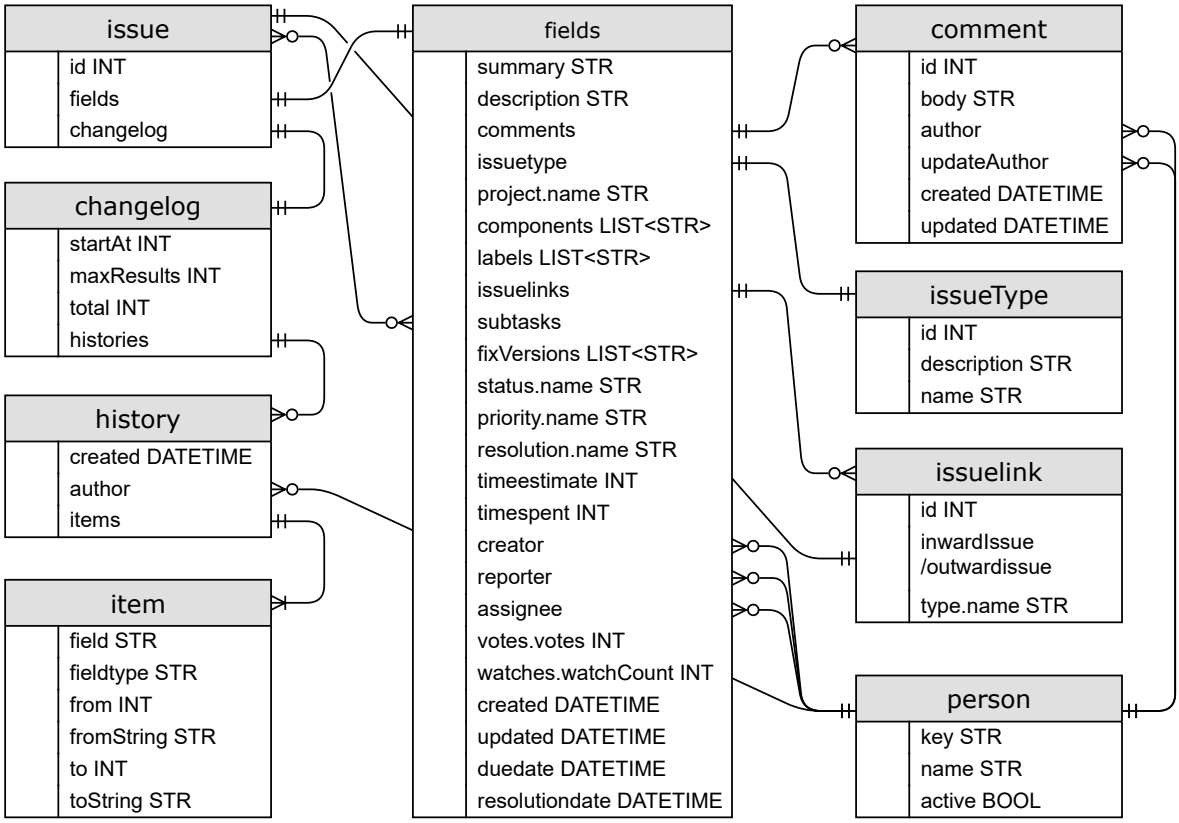


Figure 4.1.: Jira MongoDB database scheme.

each. The number of unique projects is particularly important because each Jira Repo contains documentation for multiple projects, for example, 657 projects for Apache. While the dataset has 16 different repositories, it represents 1,822 actual projects that can be studied. The data and code is publicly available with an open-source licence [160].

Other public datasets, such as the Bugzilla dataset provided by Lazar et al. [129], mainly include Bug Reports and other maintenance issues. Recent user stories datasets are rather small, and allow for neither a comparison between different issue types nor for a study of the issue evolutions [45, 235]. For this research, I only used 13 of the 16 available Jira repos because two of the repos (MariaDB and Mindville) contain no comments (an important issue field for the comparative analysis), and Mojang only contains bugs.

Each Jira Repo has a number of documented issue types, but only a subset are actually used in the projects. As shown in Table 4.1, 485 types are defined while only 352 are used. Notably, the median number of used issue types is 14, which is significantly higher than expected from research discussing ITSs (Bug Reports [27, 250], Feature Requests [67, 152], Tasks [56], or technical debt [19, 243]). The projects also vary in terms of the number of documented (186) and used (169) link types. The median number of used link types is 10, which is considerably more than the mainly well-researched “duplicate”, or “depends/relates” types.

Table 4.2.: Issue fields.

Issue Field	Description
summary	A brief one-line summary of the issue.
description	A detailed description of the issue.
comments	Community discussion on each issue.
issuetype	The issue purpose within the organisation.
project	The parent project to which the issue belongs.
components	Project component(s) to which this issue relates.
labels	Labels to which this issue relates.
issuelinks	A list of links to related issues.
subtasks	Sub-issues to this issue; can only be one level deep.
fixVersions	Project version in which the issue was (or will be) fixed.
status	The stage the issue is currently at in its lifecycle.
priority	The issue importance in relation to other issues.
resolution	A record of the issue's resolution, once resolved or closed.
timeestimate	Estimated amount of time required to resolve the issue.
timespent	Amount of time spent working on this issue.
creator	The person who created the issue.
reporter	The person who found/reported the issue. Defaults to the creator unless otherwise assigned.
assignee	The person responsible to resolve the issue.
votes	Number of people who want this issue addressed.
watches	Number of people watching this issue.
created	Time and date this issue was entered into Jira.
updated	Time and date this issue was last edited.
resolutionedate	Time and date this issue was resolved.
duedate	Time and date this issue is scheduled to be completed.

The Jira data is stored in MongoDB. Figure 4.1 describes the data as an Entity Relationship Diagram (ERD). It is not possible to create a full and reliable ERD for a document database such as MongoDB, unless a strict structure is enforced on all documents, and that is not the case for Jira across different repositories. Figure 4.1, therefore, is a simplification of the real data structure. I have diagrammed the key objects in the data, simplified structures where the complexity is self-explanatory, and extrapolated the data types to give an idea of what to expect. The primary documents to expect in the dataset are "issues", each unit of which represents a single issue within Jira. The two primary documents nested within each issue are "fields" and "changelog". The fields document contains the current attributes of the issue, and is described in Table 4.2. The issue field descriptions are primarily from the Jira official documentation.[7] The "changelog" stores the changes that have occurred to this issue. When an issue is changed, a new "history" is saved, where each changed attribute is stored as an "item". "Issuelinks" connect issues to each other and are stored on both linked issues. "Comments" are made by the community and form a discussion around the issue.

The dataset is available on Zenodo as a MongoDB dump file. The raw data includes a README file, the scripts used to download the data, the scripts used to produce the tables

[7] `https://confluence.atlassian.com/adminjiraserver/issue-fields-and-statuses-938847116.html`

and figures in this paper, and a licence (CC BY 4.0[8]). In principle, using this dataset is as easy as importing the MongoDB dump file and modifying the included JupyterLab notebook to explore the data. The data can also be queried using standard MongoDB queries. Figure 4.1 can be used to guide the initial exploration.

The data was downloaded using a Python script utilising the Jira API exposed by public Jira ITSs. I obtained the list of repos through a manual search for public Jira Repos on the internet. This search involved reviewing GoogleScholar results for "Jira" and Google search results discussing a public Jira Repo. With the public Jira URLs, the data was downloaded using the Jira REST API V2[9] and a Python script within a JupyterLab notebook. In total, ~50 GB of data was downloaded and stored in MongoDB. The initial data download was performed in May 2021 and the data was updated in January 2022.[10] Unfortunately, the MariaDB and Mindville Jira Repos are no longer publicly available, so their data was not updated.

4.3. Results

4.3.1. Artefacts and Activities in Issue Tracking Ecosystems

The Thematic Analysis produced a mapping between each issue type in each Jira and a unified set of themes and codes. Table 4.3 shows the complete results. Every issue type falls into exactly one theme and code. This mapping (stored as a JSON) allows one to programmatically ask questions such as "give me the 'Requirements' issues for Apache". In the following, I describe the five themes in more detail.

Requirements activities are documented in ITSs through lightweight representations [56] such as epics, user stories, and Feature Requests. Both top-down and bottom-up requirements are captured in ITSs. I found top-down epics and stories, as well as bottom-up Feature Requests [7]. This shows the breadth of requirements knowledge maintained within ITSs.

Development issue types track development activities such as what needs to be done and who is doing it. Example issue types include Task, Technical Task, Dev Task, and Sub-Task.

Maintenance issue types correspond to maintenance activities documented within ITSs through bottom-up (Bug, Defect, Incident) and top-down (Technical Debt, Documentation, QA Task) work. This includes quality assurance, legacy upgrades, and continuous integration.

User Support issues directly assist users of software systems regarding how to use the product. While Bug Reports focus on *issues with the code or product*, issues submitted to the User Support theme are focused on *the user and their use or interpretation of the code or product*. Example issue types include Support Request, Problem Ticket, IT Help, and Question.

Other issue types are not representative across the Jira repos, or could not be categorised at all. Example issue types include New Project, GitHub Integration, Fug, and Spike.

[8] `https://creativecommons.org/licenses/by/4.0/`
[9] `https://developer.atlassian.com/cloud/jira/platform/rest/v2`
[10] The Thematic Analysis was performed using the May 2021 data.

Table 4.3.: Homogenized issue types in the Jira dataset.

Issue types unified and grouped into themes (in bold)	Count	Examples of original names of the issue types
Requirements	**386,536**	
Epic	4,290	Epic, Roadmap item, Initiative
Story	20,441	User Story, Requirement, Story
New Feature	49,793	Feature, New Feature
Feature Request	20,175	Brainstorming, Feature Request
Improvement Suggestion	291,837	Suggestion, Improvement, Wish
Development	**245,457**	
Task	139,003	Task, Dev Task, Technical task
Sub-Task	93,763	Sub-Task, Dev Sub-task
Quality Assurance	9,872	Test, QA Task, Performance
Documentation	2,819	Doc API, Docs Task, Doc UI
Maintenance	**695,710**	
Bug Report	684,740	Bug, Incident, Defect, Issue
Legacy Upgrade	7,338	Dependency upgrade, Backport
Continuous Integration	3,632	Release, Build Failure, Tracker
Total Combined	1,327,703	

4.3.2. Prevalence of Artefacts and Activities

Requirements, Development, and Maintenance issues are all common, as detailed in Table 4.3. Of the 1,327,703 issues in the dataset, 386,536 (29%) are Requirements, 245,457 (19%) are Development, and 695,710 (52%) are Maintenance issues. While Table 4.3 displays the counts across all repos, it lacks a more nuanced perspective on the prevalence and usage of these issue types. For this, I investigate how the issues are used *across the projects*, since each project represents a distinct organisational unit within each Jira Repo.

Figure 4.2 shows the ratios of each issue type per project. Maintenance issues are the most popular across the projects, followed by Requirements and Development. For half of the studied projects, at least a third of the issues are Requirements (median 34%). This suggests that different projects seem to use requirements issues differently. For 25% of the studied projects, requirements issues seem even predominant, accounting for 47–81% of all issues (upper quartile to the upper cap).[11] Overall, Improvement Suggestions are the most popular Requirement type across the projects, followed by Feature Requests. However, as Figure 4.2 shows, a significant ratio of studied projects are heavily Story-driven. In 25% of the projects, at least 40% of issues represent User Stories. Unsurprisingly, Epics (usually used to organise user stories) are rare in most projects, with a median prevalence of 2%. Development has a median prevalence of 19%, and largely consists of Tasks (12%) and Sub-Tasks (5%). Maintenance has a median prevalence of 50%, composed almost entirely of Bug Reports with a median prevalence of 48%.

[11]The replication package [160] includes all these numbers, and the box plots.

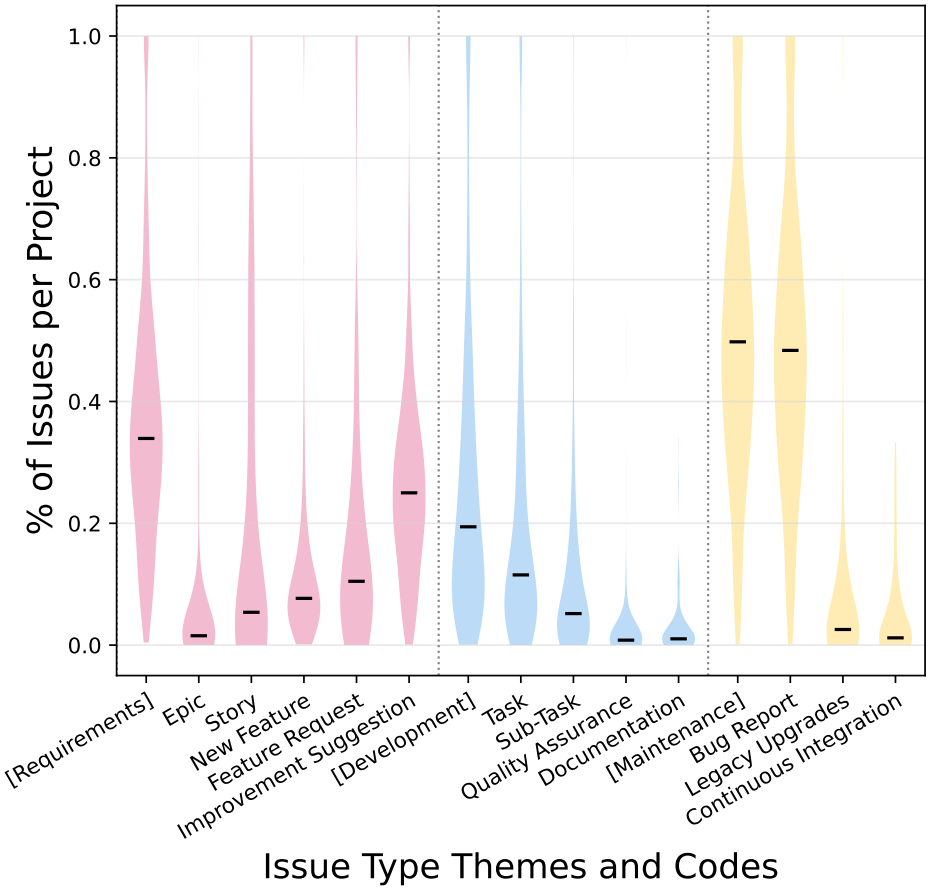


Figure 4.2.: Distribution of issue types per project.

4.3.3. Co-Occurrence of Artefacts and Activities

Jira is an ITS that has a high degree of customisability, which allows different organisations and projects to choose which artefacts and activities they use. While the above analysis describes the existence and prevalence of different activities and artefacts, it does not describe how they are used together. There are 1,477 projects within the 13 Jiras used for this analysis, so it is not reasonable to review each one in detail to get an understanding of their usage patterns. Instead, I apply co-occurrence analysis to investigate which artefacts and activities are used together. The result is a ranked list of usage pattern sets which give an idea of which artefacts and activities are being used together within these ITSs.

Traditionally, co-occurrence analysis is applied in contexts where items are grouped in various ways, such as text analysis and purchase patterns analysis. The primary goal is to surface which items commonly occur with other items, in small sets such as two, three, or four. This kind of analysis can be further complicated by considering time (which items come first), as well as magnitude (how many items at a time, or cost of the items). Instead of considering small sets such as two, three, or four, I decided to analyse the complete usage sets of artefacts and

activities. That means, for each project, I consider all artefacts and activities being used and compare those to other projects with identical or similar usage sets. This enables the analysis of the full usage across all artefacts and activities.

The "usage patterns of artefacts and activities", defined by which issue types are present in each project, is not immediately clear from the raw data. For example, what constitutes "usage" of an issue type? Clearly, the existence of issues with a certain issue type counts as usage, but how many times must a project use an issue type before it should be counted as part of the usage pattern? If a project opens a single Continuous Integration ticket to experiment, or two Sub-Tasks to see if it fits their workflow, or a User Story thinking it is a Task, does this count as part of the usage pattern? In this work, I define "usage" as using an issue type at least 5 times, or at least 10% of all issues (for projects with less than 50 issues). My assertion is that after 5 it is very unlikely to be a mistake or a small test. This reduces the total number of co-occurrence usage patterns from 215 to 186, so it does have an impact on the results, but not a considerable one. The unfiltered results (215 sets) are available in the replication package [160].

Table 4.4 shows the top 30 results of the issue type usage analysis across the entire dataset of 13 Jira repos. On the right-hand side of the Table, there are the three ITS activities and the 12 types of artefacts grouped under them. On the left-hand side, there are three project distribution values: the count of projects (1,477 projects in total), the % of projects (count/1,477), and the cumulative sum of the % of projects up to that point. The far-left value is the set number used for referencing the individual co-occurrences. To walk through an example, in set 5 of Table 4.4, only "BR" (Bug Report) is present, and all other columns are left blank. This represents the rather common usage pattern of ITSs, which is to use it *solely* as a "Bug Tracker". 86 projects in this dataset follow this usage pattern, which represents 5.82% of all projects (total of 1,477). If you add up the % of all rows above and including set 5, you get 39.61%, which represents the total percentage of projects that follow a usage pattern in set 5 or above. In other words, the top 5 usage patterns are used by ~40% of projects.

There are 186 unique usage patterns across the 1,477 projects, where 75% of all projects use only 30 different patterns (Table 4.4). The most common pattern, followed by 139 projects (~10% of all projects), is to utilise New Features, Improvement Suggestions, Tasks, and Bug Reports. This represents a sort of bottom-up RE, simple task breakdown, and documenting bugs. The second most common pattern, followed by 130 projects (~9%), is to only utilise Improvement Suggestions and Bug Reports. The third most common pattern, followed by 118 projects (~8%), is the same as the most common but with the addition of Sub-Tasks. Across these three most common usage patterns, it already applies to ~25% of all projects in this dataset. This shows a strong conformance to particular usage patterns. Additionally, there are 186 unique usage patterns with many projects with completely custom usage patterns only utilised by one or two projects. This shows a diversity of usage patterns.

These results highlight the context-sensitive nature of ITSs: there are many unique ways they are configured and utilised, even within a single ITS such as Jira. Recommendations for

Table 4.4.: Top 30 issue type co-occurrences

	Project Distribution			Requirements					Development				Maintenance		
Set	Count	%	Sum %	Ep	St	NF	FR	IS	Ta	SB	QA	Do	BR	LU	CI
1	139	9.41	9.41			NF		IS	Ta				BR		
2	130	8.80	18.21					IS					BR		
3	118	7.99	26.20			NF		IS	Ta	SB			BR		
4	112	7.58	33.78			NF		IS	Ta	SB	QA		BR		
5	86	5.82	39.61										BR		
6	71	4.81	44.41			NF		IS					BR		
7	55	3.72	48.14					IS	Ta				BR		
8	44	2.98	51.12						Ta						
9	38	2.57	53.69			NF		IS	Ta				BR	LU	
10	36	2.44	56.13						Ta				BR		
11	29	1.96	58.09					IS	Ta	SB			BR		
12	21	1.42	59.51				FR		Ta				BR		
13	20	1.35	60.87					IS		SB			BR		
14	18	1.22	62.09			NF		IS	Ta		QA		BR		
15	17	1.15	63.24		St				Ta	SB			BR		
16	17	1.15	64.39				FR	IS	Ta				BR		
17	16	1.08	65.47					IS							
18	16	1.08	66.55				FR	IS	Ta	SB			BR		
19	15	1.02	67.57	Ep	St				Ta	SB			BR		
20	12	0.81	68.38	Ep		NF		IS	Ta				BR		
21	12	0.81	69.19				FR		Ta	SB			BR		
22	11	0.74	69.94			NF		IS		SB			BR		
23	11	0.74	70.68			NF									
24	10	0.68	71.36			NF							BR		
25	9	0.61	71.97			NF			Ta				BR		
26	9	0.61	72.58	Ep		NF		IS	Ta	SB			BR		
27	9	0.61	73.19		St										
28	9	0.61	73.80		St								BR		
29	8	0.54	74.34			NF		IS	Ta	SB	QA		BR	LU	
30	8	0.54	74.88		St			IS	Ta				BR		

Requirements: (Ep) Epic (St) Story (NF) New Feature (FR) Feature Request (IS) Improvement Suggestion.
Development: (Ta) Task (SB) Sub-Task (QA) Quality Assurance (Do) Documentation.
Maintenance: (BR) Bug Report (LU) Legacy Upgrades (CI) Continuous Integration.

how to use ITSs need to be aware of these different usage patterns, rather than assume that ITSs are a universal tool with universal usage patterns. Even in the top 30 usage patterns presented in Table 4.4, there are projects that only use Requirements (rows 17, 23, and 27), only use Development (row 8), and only use Maintenance (row 5). Therefore, researchers—and practitioners—should consider the importance of understanding their data and check their assumptions before applying analyses or solutions designed to support ITS usage.

4.4. Related Work

ITSs have been studied from particular perspectives such as Bug Reports or Feature Requests, while a full understanding of this ecosystem remains rather limited. Researchers started study-

ing requirements engineering in agile settings [24, 136, 147]. However, these studies have focused on the perceptions of practitioners using interviews, surveys, and field observations. My work is complementary as it uses data analysis methods and focuses on different requirements types in addition to user stories. Recently, work by van Can and Dalpiaz [233, 234] investigated the existence of requirements in ITSs. They conducted a manual content analysis of 1,636 issues, manually labelling them as different types of requirements. Their analysis was focused on solely requirements, and was a small set of 1,636 issues. They did, however, go into much greater detail with these issues, looking at the text in each description to understand the true nature of the artefact they were labelling. As such, their analysis is complementary to mine, as their findings speak to the nature of assumed issue types and actual issue descriptions. My analysis assumes the validity of the issue types, to make broader analyses and findings across more data. Together, more generalised claims can be made about the true nature of artefacts in large ITSs.

4.5. Discussion

Our results show that, while Bug Reports and Feature Requests are indeed prominent in ITSs across projects, there are other prominent and interesting issue types that have a notable impact on how ITSs are used. Epics and Stories, for example, show prominence across projects. This implies a varied usage of ITSs, depending on which issue types are being utilised. When approaching an organisation's ITS, one should first understand how they are utilising it, before proposing various recommender systems. This includes systems such as machine learning techniques to categorise Bug Reports vs Feature Requests, or quality control of "requirements", both of which require an understanding of the extent their ITS utilises Stories vs Feature Requests. There is also a high prevalence of a diverse set of Requirements issue types (representing many styles and phases of Requirements Engineering), while Development and Maintenance issue types are quite homogenous. Development issues are mostly Task and Sub-Tasks, representing work to be done, and Maintenance issues are almost entirely Bug Reports. This highlights a rather underrepresented aspect of ITS analysis and support in both research and practice: ITSs as a primary tool for Requirements Engineering processes and documentation. Future work should further dissect these diverse issue types and seek to understand patterns of ITS usage, using the findings of prevalence and evolution as a foundation.

4.6. Summary

While SE literature has repeatedly studied Bug Reports and Feature Requests, I investigated the prevalence of various issue types representing requirements, development, and maintenance. I performed this analysis on 1.3 million issues from 16 distinct organisations across 1,477 OSS projects. The results highlight that Bug Reports and Feature Requests are indeed pervasive, but so are Stories, Improvement Suggestions, and Development Tasks. Requirements account

for 34% of all issues in a project (median across all projects), Development accounts for 19% of all issues, and Maintenance is 50%. The co-occurrence analysis revealed that 51% of projects use only eight different sets of artefacts and activities. The most popular configuration is to use New Features, Improvement Suggestions, Tasks, and Bug Reports. The second most popular configuration uses only Improvements Suggestions and Bug Reports. While the majority use only eight configurations, there are 186 usage sets across the 1,477 projects.

These results show the diversity and complexity that exists in ITSs on an artefact and activity level. However, there are other levels of information and complexity to be investigated. For example, what information exists *within* these artefacts? How often—and in what way—does this information evolve? This additional layer of complexity was already discussed by our participants in Chapter 3. They discussed problems with missing information (P1), workflow information (P6–P10), scoping issue information (P12), information islands (P14), and cumbersome issue linking (P16). I investigate these problems in further detail in the following chapter. Additionally, the findings from this chapter further highlight the diversity of information within ITSs. This diversity means additional emphasis on understanding and prescribing solutions that are context-aware, and therefore more likely to address the underlying problems. I propose a context-aware solution in Part III, beginning in Chapter 6.

Chapter 5.

Information Evolution within Issue Tracking Ecosystems

> The only constant in life is change.
>
> Heraclitus

In this chapter, I investigate the types of information found in ITSs, as well as the extent to which these information types evolve across the dimensions of frequency, time, and ownership. ITSs have become a central place for SE processes to be conducted and documented, leading to ecosystems as diverse and complex as the SE process itself. While certain dimensions of ITSs have been the attention of intense study [27, 250], the entirety of information within ITSs is not yet well understood. ITSs focus on communication [26] and iteration [98, 209], with per-issue comments [12] and community contributions. The iterative community effort is a key feature that involves the evolution of issues. Most ITSs keep track of the changes made to their issues, enabling both quantitative and qualitative investigation. Despite the rich data source that is ITS evolution data, it has yet to be fully explored.

To investigate the information within ITEs and how that information evolves, I conducted an exploratory investigation of 13 Jira ITSs using the empirical method: historical data analysis. I applied Thematic Analysis to over 20,000 issue field samples to uncover and organise the information within ITSs. I then use descriptive statistics applied to various cross-sections of the data, also in combination with the artefact types from Chapter 4, to quantify and describe the evolution occurring across the information types. The results of the Thematic Analysis revealed five issue information themes: Content, MetaContent, RepoStructure, Workflow, and Community. There are 20 codes under those themes that represent information available in issues, such as Summary and Description for the Content theme.

As a result of these investigations, we now know the distributions of major artefact types in ITEs, the involvement of different stakeholder groups, and the distribution of evolutions across many dimensions of ITEs. For each of the findings, future researchers can go into great detail within the replication package [160], which contains the full data and scripts, as well as other pre-compiled figures and findings that didn't make it into this thesis. We now have

a grounded understanding of the usage of ITEs, with more questions to be further explored by future researchers interested in particular facets of this data and context. The findings of this chapter, in combination with the industrial perspective on ITE challenges (Chapter 3) and our findings regarding requirements in ITEs (Chapter 4), present a unique opportunity to address these challenges with a theory-based solution. The next chapter will propose such a theory, with a focus on addressing these challenges and supporting future investigation of ITE problems and complexities.

5.1. Research Methodology

My primary objective with this chapter is to explore and understand evolving issue information used within ITE. To this end, I define three research questions:

RQ1 What issue information is evolving?

RQ2 How does issue information evolve?

- **RQ2A** How frequently does issue information evolve?
- **RQ2B** When does issue information evolve?
- **RQ2C** Who is evolving issue information?

RQ3 What textual changes are occurring when Requirements evolve?

I define "issue information" as the different fields that exist in issues, within a given ITS. For example, a standard issue in any ITS has the issue fields Summary and Description. Additional common issue fields include the Creator, Assignee, Labels, Priority, and Status. To investigate the issue information contained within the Jira dataset, I performed a Thematic Analysis of all available issue fields across the repositories. This process is described in detail in Section 5.1.2.

To investigate how issue information evolves, I analysed the evolution history of issues. I was interested in three primary perspectives: frequency, time, and ownership. I performed a number of cross-analyses to understand how evolution occurs across subdivisions of the data, such as the issue information types, as well as the issue artefacts and activities from Chapter 4.

5.1.1. ITS Dataset

To address these research questions, I started with the same dataset collected and described in Chapter 4. The full dataset contains 16 public Jira repos containing 1,822 projects and 2.7 million issues. For this chapter, I only used 13 of the 16 available Jira repos because two of the original repos (MariaDB and Mindville) contain no comments (an important issue field for the comparative analysis), and Mojang only contains bugs.

I cleaned the data by removing issues that are not yet resolved, as well as unexplainable and unusable data (e.g., issue created date is null). Jira also stores the fields entered at the

Table 5.1.: Research data: 13 public Jira repositories.

Jira	Born	Projects	Issues	Evolutions	Authors
Apache	2000	892	724,226	7,663,616	96,368
Hyperledger	2016	36	19,544	176,354	1,740
IntelDAOS	2015	6	6,134	73,554	112
JFrog	2006	32	8,001	62,097	2,493
Jira	2002	143	134,297	1,249,338	69,329
JiraEcosystem	2004	152	28,289	256,298	3,851
MongoDB	2009	96	62,370	607,296	11,698
Qt	2003	44	79,480	779,291	17,614
RedHat	2001	504	180,890	1,609,278	16,171
Sakai	2004	50	24,376	217,310	1,330
SecondLife	2009	16	451	6,729	193
Sonatype	2008	28	4,893	52,787	1,681
Spring	2003	95	54,752	491,818	14,549
Sum		2,094	1,327,703	13,245,766	237,129
Median		50	28,289	256,298	3,851
Std Dev		245	187,035	1,976,001	28,712

issue creation time in the evolution data. When an issue is created, ~12 fields are filled in on average (mean), often with the default value of null. I exclude these "creational" evolutions from my analysis to focus on issue evolution, not issue creation. As a result, I removed 16,518,405 creational data points from the dataset, leaving 13,245,766 evolutions of issues fields.

Table 5.1 shows an overview of the research data for this investigation. For each repo, it lists the creation year, the number of unique projects, issues, evolutions, and evolution authors. The research data consists of 13 Jira repos, 2,094 projects, 1.3 million issues, 13 million post-creational evolutions, and 237k authors. The count of 2,094 projects is higher than the 1,822 projects reported in Chapter 4 due to the difference in how the projects are counted: Chapter 4 reports unique *final* projects, whereas here I am reporting all unique projects ever assigned, which is found in the evolution history. Chapter 4 does not analyse the evolution of issues, and therefore does not consider these intermediate values. The data and scripts are available in my replication package [160]. The package includes additional information in the form of box plots, medians and quartiles, and the results of various additional investigations.

5.1.2. Thematic Analysis

My goal was to produce a set of generalised issue fields across the Jira Repos, and group them into information themes for further analysis. Unifying the issue fields and forming the information themes was a non-trivial process due to the 2,562 unique issue fields in the dataset. Despite the large number of unique issue field names, there is still a standard set of fields that most Jira Repo utilise, and many of these unique issue field names are synonyms. Therefore, I sought to unify these fields into a smaller set of summarised fields. To address these issues, I

conducted a Thematic Analysis [33, 43] of issue fields across the 13 Jira repos. I defined two analysis criteria for the investigation: 1) each field must exist in all the Jira repos, and 2) each field must evolve. The first criterion is to obtain a *generalised* set of issue fields: they must exist in all studied repositories. The second criterion is to obtain *evolving* issue fields: they must all evolve (not all fields evolve with the dataset).

Thematic Analysis is characterised by initial decisions: the type of analysis, the style of approach, and the level of analysis [33]. For the type of analysis, I chose a Rich Overview. The goal was to get an overall understanding of the issue information contained within ITSs, which requires all facets of the data to be considered. For the style of approach, I selected an Inductive approach. I was interested in unifying and grouping the issue information given the emergent ideas I would find by looking closely at each of the issue fields. For the level of analysis, I selected Semantic Themes. I was interested in understanding the surface information presented when viewing examples of these fields in the data, without interpreting beyond what was described. In summary, I conducted an Inductive Thematic Analysis, providing a Rich Overview that produced Semantic themes. Thematic Analysis involves five main phases, presented here.

In phase 1, I familiarised myself with the data by looking through issue field samples. For each issue field, I looked in three locations: 1) the final state of the fields in issues, 2) the history of the fields in issues, and 3) the documentation provided by each Jira Repo that explains the field. I used the first two locations to construct a full history of the field throughout the lifetime of the issue.[1] The third location helped disambiguate field names and usage across Jira Repos. I utilised this familiarisation process to already begin systematically searching for issue fields that satisfied the two analysis criteria. I found that there were a fair number of issue fields that shared the same names, and even more issue fields that were close in name, but not identical. Inspecting examples of these issue fields, it became clear that some organisations had simply chosen to rename the fields, even though they were using the issue fields for the original purpose.

In phase 2, I generated an initial set of codes. Within the 13 Jira Repos, there are 2,562 unique field names across the three studied locations (union), and only 22 unique field names that exist in all three of these locations (intersection). When adding the restriction that each field name must exist in all 13 Jira Repos, there are then only 10 unique field names across the three locations.[2] My goal in applying the analysis criteria, however, was to go beyond exact field names and look for different or modified names given the non-standard nature of Jira Repos. Here is an example of why this is necessary: issue "links" are an important field within Jira, but they are called "issuelinks" in the issues and "Link" in the histories. Without this analysis, issue links would have been missed. I used a script to randomly select issues to investigate, and recorded this process in an output file. I iteratively inspected (and documented) 20,374 issue field samples across the three distinct issue field locations. 20 issue fields satisfied the analysis criteria, and formed the final set of codes for the Thematic Analysis.

[1] The history alone is not sufficient, since fields that are set but never changed are not listed in the history.
[2] Exact details of these fields available in the replication package [160].

In phase 3, I re-focused the analysis on the broader level of themes across the 20 codes from phase 2. Using the 20,374 field samples, I investigated the way in which these fields were being used within each Jira Repo. I built the initial set of themes based on how the issue fields were being used to support issues within the organisational purpose of Jira. For example, some issue fields such as IssueType and Parent were clearly being used to provide structure to the issues within Jira; fields such as Summary and Description were the core content of the issues; and fields such as Status and Priority were being used to support the workflow of issues. The result of this phase was an initial set of themes with all codes mapped underneath.

In phase 4, I challenged the themes by reviewing them with four other researchers in the field, iterating over them, and checking additional data points. I had some movement within the themes, e.g., Comments was as a controversial field because it contains content-like information, but ultimately can be used to communicate anything and therefore fit much better under Community. The result of this phase was a final set of themes.

Finally, **in phase 5**, I defined each theme and verified that each issue field matched that definition, forming a final set of five themes that contain 20 codes with sufficient mutual exclusion. I discuss each theme and code in detail in the following subsections. Much of our understanding of the fields comes from the Thematic Analysis, while some information was gained through official Jira documentation.[3]

5.1.3. Evolution Analysis

To quantify the extent to which evolution is occurring within ITSs, I compare how the issue information and the SE artefacts and activities collected in Chapter 4 evolve across a generalised set of ITSs. This process is exploratory in nature, and thus involves observing, understanding, and interpreting real-world data. While interviews and content analysis provide much value in contextualising detailed and rich information, they do not allow for an overview over large amounts of data, particularly when looking to understand cross-organisation behaviour. For this, I chose the "Sample Study" strategy as described by Stol and Fitzgerald [216]. This strategy involves either the collection of data through mass questionnaires, or finding large datasets through data mining. I selected data mining and performed a historical data analysis of the large dataset. This enabled me to investigate issue evolution across many contexts, and provide generalised evidence towards the exploratory objective. With these techniques, I conducted an empirical study of 13 Jira ITSs, across 1.3 million issues with 13 million evolutions. I investigated the frequency of the evolutions, the timing (relative to the issue creation), and the ownership (who changed the field). For the description field, I also performed a manual analysis of a random sample to understand what was changing syntactically and semantically.

Before performing the analyses, I first created an evolution dataset from the raw Jira data. By

[3] Jira docs on issue fields: `https://confluence.atlassian.com/adminjiraserver/issue-fields-and-statuses-938847116.html` and at `https://confluence.atlassian.com/jirasoftwareserver/advanced-searching-fields-reference-939938743.html`

default, Jira data is stored in a JSON format as fully detailed with an ERD in Figure 4.1. While the JSON format is easy to interface with, and has advantages for certain types of selective queries, it is not an ideal format for queries that need to look across the entire dataset. For this, I created a single Pandas DataFrame[4] that stores the data in an SQL-like format that is quick to perform full dataset queries. The process of weaving together the evolution history was not a trivial process, given the rather unorthodox way in which Jira stores the history of each issue. Additionally, different fields are stored in different ways within the history. The resulting DataFrame and scripts to produce the DataFrame are available in my thesis replication package [160]. The evolution dataset has 30 million evolutions from the 1.3 million issues. Of the 30 million evolutions, 17 million evolutions (56%) are the "creational" evolutions: records of fields that were set when the issue was created. This leaves 13 million evolutions (44%) that are post-creational evolutions: records of fields that were changed after the issue was created. For the entire analysis of RQ2 and RQ3, "evolution" refers to "post-creational evolution".

I use box plots to present the results for RQ2A and RQ2B. Box plots summarise a single dimension of data (list of values), visualising the median, the upper- and lower-quartiles (edges of the box), and the upper- and lower-whiskers (the lines drawn at each end of each line). The median (middle value in a sorted dataset) is resistant to outliers, which is important for visualising the results.[5] Each box represents the Interquartile Range (IQR), where the top and the bottom of the box represents the median of the upper and lower halves of the data (split on the original median). In other words, the IQR (box) represents the middle 50% of the data: the most expected values. Finally, the upper- and lower-whiskers represent the highest and lowest values within the dataset, to a maximum of 1.5x the IQR. This is the popular choice[6] for visualising the whiskers, and is the default option of the Python visualisation library matplotlib.[7] I do not visualise outliers beyond the whiskers.[8]

I split the data into Requirements, Maintenance, and Development issues. Each box plot figure visualises sets of three box plots, where the orange box (the first box) represents the Requirements issues, followed by the Maintenance issues, and then the Development issues. Each box plot axis is labelled with their respective data scale, which varies from frequencies to distributions, and normal scale to (sym-)log scale.[9] All scripts used to format the data and produce the figures are available in the replication package [160].

[4] `https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html`
[5] ITSs have inexplicably strange outliers, a great subject for future research.
[6] `https://en.wikipedia.org/wiki/Box_plot`
[7] `https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html`
[8] To see all data points, see the replication package [160], and turn on outliers.
[9] The "Symlog" scale (symmetrical log) allows values between 0 and 1, and is used in the analysis so that values close to 0 can be interpreted and visualised.

5.1.4. Issue Content Analysis

To address RQ3, I conducted an open-coding content analysis to investigate requirements evolution patterns in the content of issues. I performed an exploratory open-coding of 156 issues (78 each) with one or more evolutions to the description (for a total of 315 evolutions) across all Jira Repos. I stratified the random selection across the 13 Jira Repos and the three issue types (Requirements, Maintenance, and Development), selecting four issues from each, thereby arriving at $156 = 13 \times 3 \times 4$. The goal of this process was to gain a preliminary understanding of content changes occurring across all Jira Repos and issue types and to identify change recurring reasons, which I call content evolution patterns. Given the goal of a "preliminary understanding", I did not aim for representativeness or concept saturation, but rather diversity. That being said, I did find by the end of the open-coding process that the patterns were saturated, although I did not design or track this systematically. The aim was to explore the existence of patterns rather than quantifying them. Once the task was complete, I met to with Prof. Dr. Maalej to discuss, combine, and agree on the patterns found. Overall, we found a similar set of patterns but with different frequency and with slightly different names. Discussing the findings based on examples led to a shared and uniform pattern list. The full pattern extraction data, including labelled samples, is included in the replication package [160].

5.2. Results: Information Types

The results of the Thematic Analysis applied to the issue fields reveals a rich ecosystem of information that covers many aspects of the SE lifecycle. Here, I present the themes and codes that emerged from this analysis. Table 5.2 shows the themes, codes, and issue field names that emerged from this analysis. There are five issue information themes: Issue Content, Issue MetaContent, Issue RepoStructure, Issue Workflow, and Issue Community.

5.2.1. Issue Content

The Issue Content theme covers issue fields that form the core information of an issue. The two codes in this theme are the *Summary* and the *Description*.

The **Summary** field acts as a title for the issue, providing a brief overview of the information contained in the issue. For example: "Allow toggling every single projector's ability to cast a shadow separately" is a summary of STORM-2147 from SecondLife[10]. Arguably, the Summary is the most important field for representing the issue. Important tasks in ITSs include searching for issues, prioritising issues, assigning issues to people, and managing the backlog. These tasks will likely be done by only reading the Summary field. As such, a well-written Summary should touch on all important high-level aspects of the issue. There was only a single issue field name for Summary found during the analysis, which is "summary". This perfect alignment across the

[10] `https://jira.secondlife.com/browse/STORM-2147`

Table 5.2.: Information themes and codes, and issue field names.

Themes and Codes	Associated Field Names
Issue Content	
Summary	summary
Description	description
Issue MetaContent	
Labels	labels, Labels
Environment	environment, Environment, Environment: other information
VersionsAffected	versions, versions.name, Version
VersionsFixed	fixVersions, fixVersions.name, Fix Version
Issue RepoStructure	
IssueType	issuetype, issuetype.name, Issue Type, type, issueType
Project	project, project.name, Project, Project Name
Components	components, components-name, Component
Parent	parent, Parent, Parent Issue, IssueParentAssociation, Parent Link
IssueLinks	issuelinks, Link
Issue Workflow	
CreatedDate	created
ResolvedDate	resolutiondate
Status	status, status.name, Status, Current Status
Priority	priority, priority-name, Priority
Resolution	resolution, resolution.name, Resolution
Issue Community	
Creator	creator, creator displayName
Reporter	reporter, reporter displayName
Assignee	assignee, assignee displayName
Comments	comment, comments, Comment

13 Jira Repos and in all three locations (issue final state, issue history, and issue documentation) is likely due to the importance of the field and the forced structure by Jira itself.

The **Description** field contains the full-text details for the issue. Arguably, the Description is the second most important field in the issue (after the Summary). When the Summary is insufficient, or the reader wants a complete understanding of the issue, they will go to the Description. Issue Descriptions come in many forms, from simple small paragraphs to multi-section deep dives. Additionally, the form of information in Descriptions is not always natural language prose, but can also include headers, code blocks, log outputs, steps-to-reproduce, and more. Accordingly, the processes of understanding, unpacking, interpreting, and presenting the results are all subject to increased complexity. In principle, all necessary information is listed in the Description, but in reality, some information is listed elsewhere in the issue, and some information is not listed at all. For example, it is common for important Description-like information to only be included in the Comments field. Same as Summary above, the Description only had a single field name across all Jira Repo and locations: “description”.

5.2.2. Issue MetaContent

The Issue MetaContent theme covers fields that enhance the Content fields. Importantly, these are distinguished from the Issue Content fields in that MetaContent information is shared across many issues (e.g., shared labels and environments), whereas Content information is unique. The four codes in this theme are the *Labels*, *Environment*, *VersionsAffected*, and *VersionsFixed*.

The **Labels** field contains tag-like information shared across issues. An example of Labels can be seen in the Bug Report MCPE-31887[11] which has multiple labels referring to the various Minecraft items that the Bug Report relates to: acacia, birch, button, oak, pick, pick-block, pickblock. These labels enable two primary functions within the ITS: meta-description and categorisation. They add meta-description in the form of at-a-glance descriptors that further add context. By looking at the labels, a reader can gain a much quicker understanding of the issue. Labels also add a deeper form of categorisation within an ITS, since these labels can be used to query the issues. For example, a developer working on the above Bug Report may want to view open Bug Reports with the "pick-block" label. This would allow them to identify related issues that may add valuable context to this Bug Report, or perhaps there is an opportunity to fix multiple Bug Reports at once if they share this underlying context.

The final three thematic codes for the Issue MetaContent theme are all label-like, except they have specific types of information within them. The **Environment** field focuses on hardware- or software-related information. The **VersionsAffected** field contains the software version(s) affected by the content described in the issue. The **VersionsFixed** field contains the software version(s) fixed by the content described in the issue.

5.2.3. Issue RepoStructure

The Issue RepoStructure theme covers fields that provide structure to Jira Repos. The four codes in this theme are the *IssueType*, *Project*, *Parent*, and *IssueLinks*.

The **IssueType** field describes the role this issue plays within the ITS. For example, the issue MCPE-31887[12] has the IssueType "BugReport", while the issue JRASERVER-7784[13] has the IssueType "Suggestion". These two types greatly distinguish how these issues are handled within these organisations. Bug Reports are issues that likely need to be addressed or else the software will not continue to function as it should. Suggestions are a form of button-up requirements that *could* be implemented, if the developers believe it enhances the software in a way that benefits the stakeholders enough. Other examples of IssueTypes includes Epic, Story, Feature Request, Work Item, and Quality Assurance. See Chapter 4 for a comprehensive breakdown of IssueTypes, and a thematic mapping into artefacts and activities.

The **Project** field is the primary top-down organisational construct used to provide structure

[11] `https://bugs.mojang.com/browse/MCPE-31887`
[12] `https://bugs.mojang.com/browse/MCPE-31887`
[13] `https://jira.atlassian.com/browse/JRASERVER-77884`

to Jira Repos. Every Jira Repo is composed of a set of projects, and every issue is organised into exactly one project. The idea of a Project is that it is mutually exclusive within the Jira Repo. The concept of a Project, however, is different across Jira Repos. Some Jira Repos have true project-like values that represent different internal products or large initiatives inside companies. Other Jira Repos use the Project field to distinguish between teams such as QA or Testing. Regardless of the type and granularity of information stored in the Project field, it is always the case that an issue can only have exactly one value assigned to the Project field. The **Components** field also acts as a sort of top-down organisational construct within Jira Repo; however, Components are not mutually exclusive like Projects, and a single issue can have multiple components assigned to it. Components are also used in diverse ways by organisations. The **Parent** field contains a link to the parent issue that this issue is organised under. The use of the Parent field naturally forms tree-like structures within the data, since every issue can only have one parent, but a parent issue can have multiple child issues. The concept of a parent is generic, and therefore can be used for many reasons.

The **IssueLinks** field is a list of links to other issues that are related in some way. See Montgomery et al. [164] for a comprehensive breakdown of issue link types. Many ITSs offer the ability to link issues together via specialised "link" types. These links relate issues to each other, allowing stakeholders to understand and capture the relationships between different issues, making it easier to find information, and structuring overall project knowledge [199]. For example, a reported issue might be part of a major bug, a bug might contribute to a specific requirement, or two issues might refer to the same topic. Issue links can also be described as *horizontal traceability* as they relate artefacts (issue reports) on the same level [85, 99]. Issue links (and their types if used in the respective tool) are listed either on the corresponding issue page (e.g. JIRA, Bugzilla) or in the comments sections (e.g. GitHub Issues). There are different link types, such as Duplicate, Relate, Block, and Subtask, which serve different purposes and can be used to indicate various relationships between issues.

5.2.4. Issue Workflow

The Workflow theme covers fields that support issues in their lifecycle from creation to resolution. The five codes are the *Status*, *Priority*, *Resolution*, *CreatedDate*, and *ResolvedDate*.

The **Status** field marks the current stage of the lifecycle an issue is in. Organisations, projects, and teams define workflows for different types of issues, where each workflow consists of stages an issue must go through from "Opened" to "Closed". The Status field stores the current stage of the issue. The collective set of workflow stages forms a set of nodes in a potential workflow diagram. To get the actual workflow diagram for a given organisation, the history of issues needs to be investigated to understand the order the Statuses are normally used. The **Priority** field records the importance of an issue in relation to other issues. The **Resolution** field marks the reason for closing an issue. The **CreatedDate** and **ResolvedDate**

mark the beginning and end of an issue lifecycle. However, the CreatedDate cannot change, and any changes to the ResolvedDate are not tracked in the evolutions. I kept these two fields in the analysis because they enable time-related analysis (particularly RQ2B). Individual use cases need to investigate the value of the minimal natural language content stored in these fields. The only field name found during the analysis is: "created". The **ResolvedDate** field records the date the issue was resolved.

5.2.5. Issue Community

The Community theme covers fields that describe the community of stakeholders and their involvement in the issue. The four codes are the *Creator*, *Reporter*, *Assignee*, and *Comments*.

Issue Community covers fields that describe the community of stakeholders and their involvement in the evolution of an issue. The **Creator** field lists the person who opened the issue in Jira. The **Reporter** field contains the person who initially brought up the issue (which is before issue creation). The **Assignee** field lists the person responsible for the issue. The **Comments** field lists all the comments made on this issue. For example, "That could be fun to implement, although I won't have time for it for a while" from a comment on QTBUG-76315,[14] where a developer conveys interest but also time constraints which led to this issue not being done, despite being 4 years old. Another example would be "I think you are right and [because] three months have passed, I can fix it for you. Thank you for the issue and the suggested code changes." on ZOOKEEPER-4703[15] In most ITSs, the comments are displayed in ascending chronological order after the Description. Each comment tends to be just a few sentences long, but it can also be multiple paragraphs. The content of Comments includes questions, personal updates, requests for information, and additional information. The comments may contain relevant information not included in the Description. For example, a comment might contain the log output of the error reported in the description. This can happen as ITS maintainers comment and request additional information, and the original issue reporter replies with the requested information instead of updating the original Description. The worst form of this is when conflicting information is listed in the comments, or when information specifically designed for other fields is listed in the comments (such as the Status field). The consequence of this spreading of information is that all future readers must consume all comments to understand the issue. A complete interpretation requires analyses or scripts that align the information listed in all three of the Summary, Description, and Comments fields.

[14] `https://bugreports.qt.io/browse/QTBUG-76315`
[15] `https://issues.apache.org/jira/browse/ZOOKEEPER-4703`

5.3. Results: Information Evolution

5.3.1. Frequency of Issue Evolutions

To answer RQ2A, I counted the number of evolutions to each issue, and broke that down into issue types and information types. I present the results of this analysis in Figures 5.1 and 5.2.

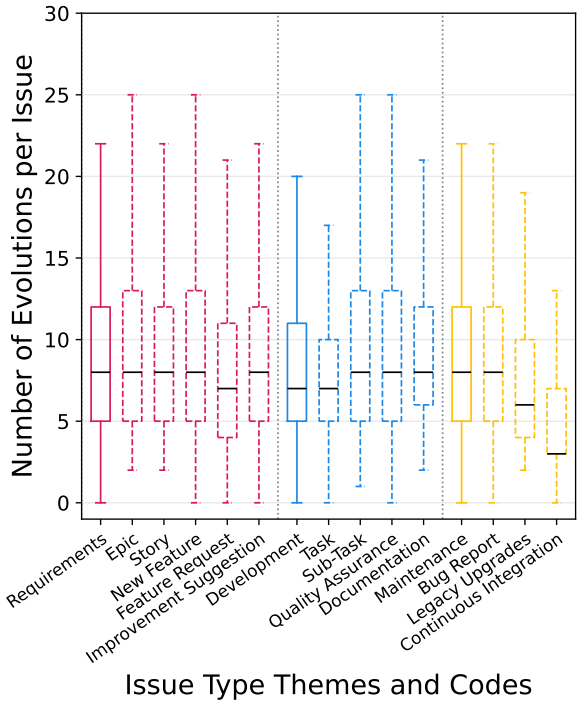


Figure 5.1.: Number of evolutions per issue type.

Requirements evolve a median of ~8 times in their lifetime, with an IQR of 5–12 evolutions. This is the same for Maintenance issues, while Development issues evolve slightly less. Epics and New Features have about 5 more evolutions than the rest of the requirements types (up to 25 in total), which is the same for Task and Sub-Task for Development. I visualise the number of evolutions per issue type in Figure 5.1. Note that all Epics and Stories evolve at least two times while all other requirements types have some issues that did not evolve at all (lower cap goes to 0). All issues in the dataset are resolved, so an issue with zero post-creational evolutions was created in an already-resolved state. This suggests that Epics and Stories go through a more structured process, requiring at least some interaction before resolution.

Looking at the distribution of evolutions across the information themes (Fig. 5.2), we see that *Workflow* evolutions occur the most, at ~40% (median) of all evolutions on Requirements issues, while Content evolutions occur the least, with a median of 0% and an upper-quartile of 6%. Both Development and Maintenance share similar distributions, although there are some differences—particularly in Community evolutions. For *Community* evolutions, Epics have the

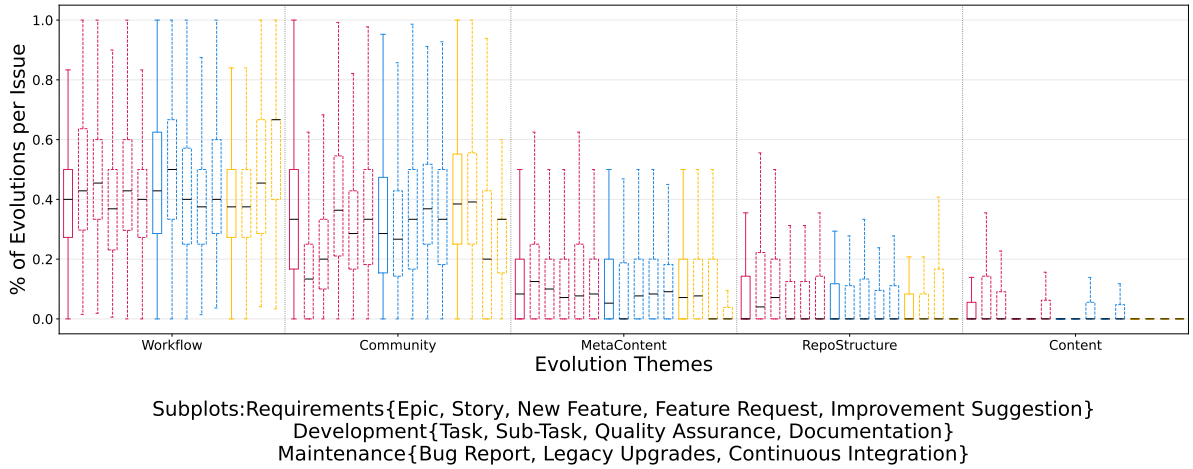


Figure 5.2.: Distribution of evolutions across issue information themes, per issue type.

least percentage of evolutions at 13%, similarly low as Stories at 20%. This highlights that Epics and Stories have fewer discussions or changes of ownership (or both) than other types of issues. This is in contrast to *Content* evolutions, where Epics have the highest evolution percentage, followed second by Stories. In other words, after creation, issue Content barely changes for all issue types, except sometimes for Epics and Stories.

5.3.2. Issue Evolution Time

On average, Requirements continue to evolve 27 days after their creation (median), compared to Development at 11 days and Maintenance at 8 days. Epics continue to evolve the longest of all Requirements types at 81 days, while Stories evolve the shortest at just 22 days. Although Stories evolve the shortest of all Requirements types, this is still double that of Development and almost triple that of Maintenance. I visualise these results in Figure 5.3 by plotting the time-of-change offset from issue creation (in days), for each evolution in each of their respective subgroups.[16] The upper quartiles show an even larger difference with Requirements at 260 days, compared to Development at 72 days and Maintenance at 94 days. In summary, Requirements issues evolve 2–3x longer than the other issues. Even though all issue types tend to evolve the same number of times (~8 times), Requirements evolutions need 2–3 times a longer period after creation, with Epics being even more exaggerated at 7–10x longer than non-Requirements.

When breaking down Requirements into information themes (Fig. 5.4), I see that RepoStructure continues to evolve 118 days later, compared to MetaContent at 49 days, Workflow at 36 days, Community at 16 days, and Content at just 1 day. In other words, while structural information such as project, component, and links continued to be organised within the

[16]Note the y-axis is in log scale, which makes these values appear visually closer than they are. Technically, in Symmetrical Log, to plot values between 0 and 1: `https://en.wikipedia.org/wiki/Logarithmic_scale#Extensions`.

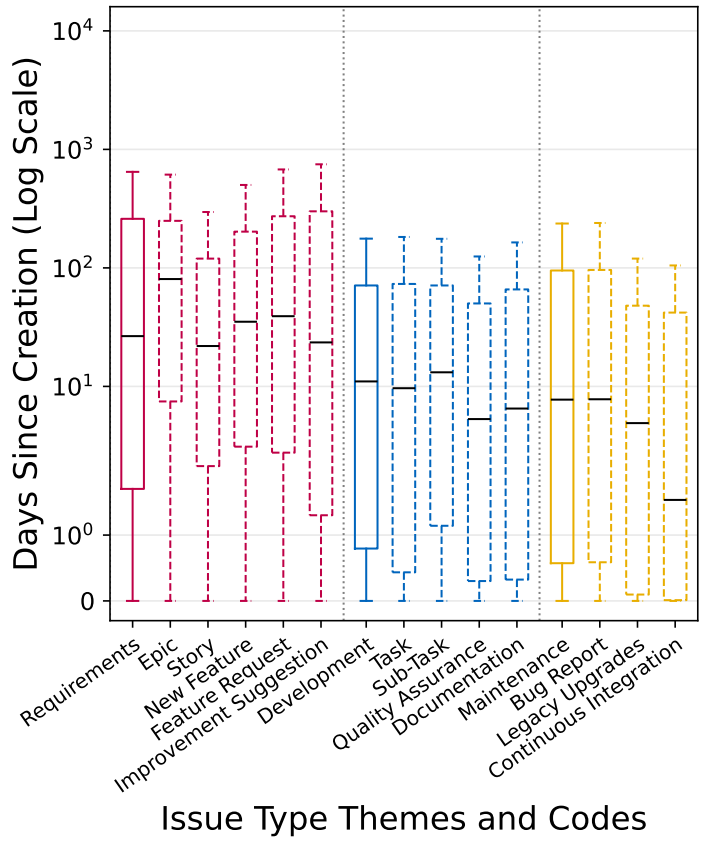


Figure 5.3.: Evolution time after issue creation, per issue type.

ITSs for months (sometimes years), 50% of the Content evolutions occurred immediately after creation. Looking at RepoStructure, Requirements continue to evolve much longer (118 days) than Development (8 days) and Maintenance (17 days). This shows a higher uncertainty to Requirements organisation than other issue types. One potential explanation is that some issues originally filed as bugs might emerge as new requirements while working on them. Another explanation is that requirements issues might simply be more complex.

Looking into the five Requirements types in dotted red in Figure 5.4, Epic Content evolutions continue to evolve much longer at a median of 8 days, compared to the Requirements overall (1 day). This suggests either a more careful editing and update of Epics or new details that emerge after opening the Epic. Epic Workflow evolutions also occur much later at 118 days compared to Requirements overall at 36 days (and even later compared to Development at 12 days and Maintenance at 11 days). This shows the need for more time when it comes to working with Epics compared to all other issue types.

When analysing the individual information fields, I see large differences between them: some fields evolve for months (medians) and even years (upper-quartiles), while other fields evolve

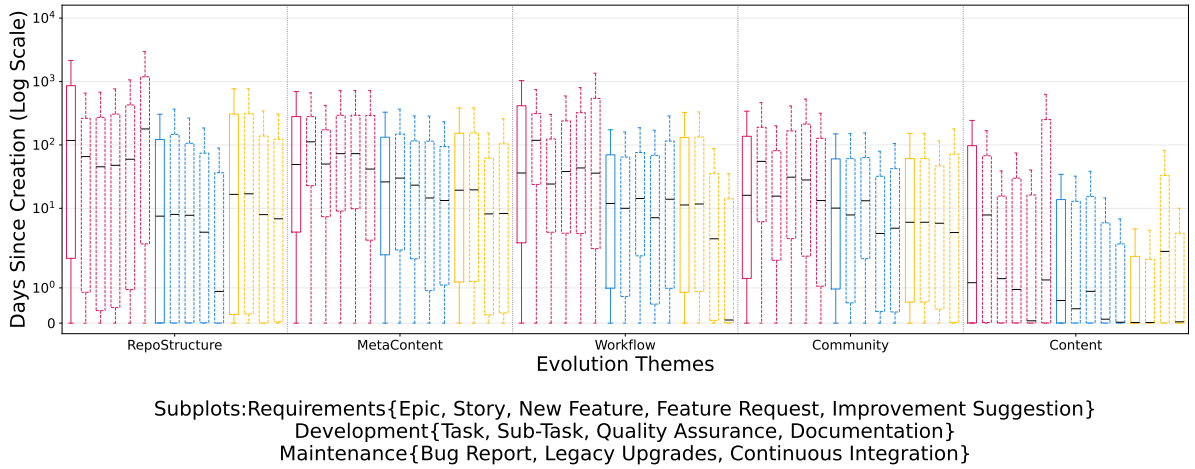


Figure 5.4.: Evolution time after issue creation, per evolution theme and issue type.

for just a few hours. I applied the Jump Method[17] to cluster the Requirements medians, which revealed three distinct magnitudes of evolution time: months, weeks, and days.[18] I present these findings in Figure 5.5 where the three distinct magnitudes are visualised as horizontal dotted blue lines representing the mean of the Requirements medians in each magnitude group. The first magnitude, consisting of seven fields, evolves for an average of 7 months (212 days). The second magnitude also contains seven fields and evolves for an average of 4 weeks (30 days). The final magnitude contains just three fields and evolves for an average of 22 hours. There are many notable differences between Requirements and the other types. IssueType is the most extreme, with Requirements IssueType evolutions occurring a median of 460 days after creation, while Development and Maintenance are only 7 and 6 days, respectively. Other notable differences include Parent, Labels, Priority, Reporter, Summary, and Description.

Looking into the five Requirements types in dotted red, most issue fields show a large difference between the Requirements types. Most notably, Epics and Stories are rather consistently different from the other Requirements types (and the Requirements median). For example, looking at IssueType and Parent, Epics and Stories are one magnitude higher than most other Requirements types. For Reporter, Epics and Stories evolve for a median of 1 and 27 days, respectively, compared to the Requirement's median of 89 days. Regarding Resolution, Epics have a median evolution time of 164 days, compared to the Requirement's median of 25 days. Finally, for Description, Epics evolve a median of 7 days, compared to the Requirement's average of 1 day. These are just some examples of the extreme differences visualised in Figure 5.5.

[17] Sort the data and split on largest gaps N times: `https://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set`

[18] At this level of analysis, there are technically five groups, but two non-presented groups contain just one field each, at both ends of the figure. Given the nature of largest-gap analysis applied to log-scale data, I deemed it useful to combine those two outer groups with their representative larger groups. It makes no difference to the analysis of the inner three groups regarding the grouping.

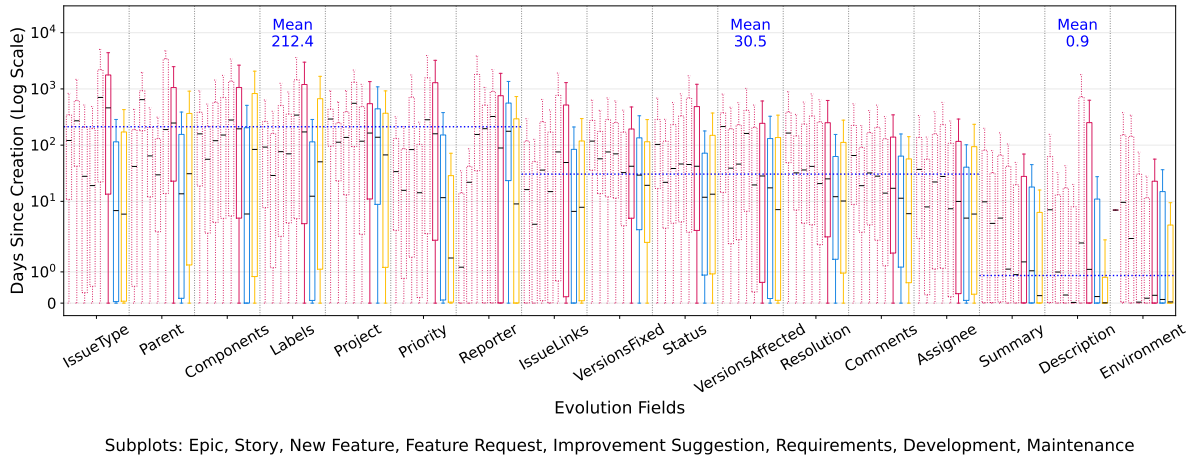


Figure 5.5.: Evolution time after issue creation across issue fields, per issue type.

5.3.3. Issue Evolution Stakeholder

Method and Visualisation

I compared the Jira-defined ownership roles against who is evolving issues, and split those observations along two primary dimensions: activities (from Chapter 4) and information types. I present the results of this analysis in Table 5.3. There are three ownership roles defined in Jira: Creator, Reporter, and Assignee. The three ownership roles are not mutually exclusive. For example, Gina can "report" a Feature Request and enter it into the system by "creating" the issue, and then she "assigns" herself to be responsible for it (Gina is now all three roles at once) [9]. It is also possible to interact with an issue even if you are not one of those roles, which therefore defines a fourth mutually exclusive role: Non-Owner. When an issue is evolved, I check and record the evolution author against the current state of the four roles. Table 5.3 visualises the distribution of evolutions performed by the respective ownership groups, described across activities and information types. The first two data rows represent the distribution of evolutions done by either an Owner (a Creator, Reporter, or Assignee), or a Non-Owner (none of Creator, Reporter, or Assignee). Each column of these two rows add up to 100%, as all evolutions are performed by either an Owner, or a Non-Owner. The next and final seven rows represent subsets of Owners, where the row names denote which subsets the evolver was in: uppercase means "within set" and lowercase means "not in set". The final seven data rows are all subsets of the Owner roles: all combinations of **C**reator, **R**eporter, and **A**ssignee.

The Owner role subsets:

- **Data-Filled Sets**
  - CRA: The evolver is all three: Creator, Reporter, and Assignee.
  - CRa: The evolver is Creator and Reporter, but not Assignee.
  - crA: The evolver is only the Assignee.

Table 5.3.: Ownership Distribution Heatmap across Evolution Types and Issue Types

Scope	**S1**	**S2**			**S3**					**S4**														
EvoType	All	All			Con	Met	Rep	Wor	Com	Con			Met			Rep			Wor			Com		
IssType	All	R	M	D	All					R	M	D	R	M	D	R	M	D	R	M	D	R	M	D
Owner	48	43	47	58	79	39	43	56	41	69	81	87	38	37	49	37	41	60	50	54	71	36	43	44
Non-Owner	52	57	53	42	21	61	57	44	59	31	19	13	62	63	51	63	59	40	50	46	29	64	57	56
CRa	37	36	40	30	64	33	42	23	46	59	73	44	34	32	35	43	41	43	22	25	19	44	50	36
CRA	34	39	27	48	29	36	36	43	25	33	20	47	38	29	49	38	29	46	48	35	56	31	19	38
crA	28	24	32	20	7	30	20	33	26	7	6	8	26	38	15	17	28	10	28	39	24	24	29	22
cRa	1	0	1	1	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	1	1	2
Cra	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1
cRA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CrA	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Col Names: **R**equirements, **M**aintenance, **D**evelopment, **Con**tent, **Met**aContent, **Rep**oStructure, **Wor**kflow, **Com**munity.
Row Names: **Evo**lution**Type**, **Issue**Type, **C**reator, **R**eporter, and **A**ssignee. Uppercase: within set, lowercase: not in set.
Cell Colours: All cells use the same heatmap intensity scale, and **R**equirements data is orange.

- **Mostly Empty Sets**
  - *Cra*: The evolver is only the Creator.
  - *CrA*: The evolver is Creater and Assignee, but not Reporter.
  - *cRa*: The evolver is only the Reporter.
  - *cRA*: The evolver is Reporter and Assignee, but not Creator.

Each column of these seven rows also adds up to 100%, but 100% of the Owner evolutions (not all evolutions). The final four rows have almost no data in them. Looking at these subsets, they represent situations where the Creator is never the same person as the Reporter. Given the low percentage of data (less than 1%), this is quite a rare occurrence. Every cell is additionally shaded with a different intensity of orange or grey, where the intensity is equal to the value of the number shown. Requirements have been visualised in orange. Table 5.3 visualises the data in four column scopes: (**S1**) all data without dimensional splits, (**S2**) split along the activities, (**S3**) split along evolution type, and (**S4**) split across evolution and activity.

Results

The evolution distribution between Owners and Others is almost equal at 48% and 52%, respectively. The bottom three rows show a similar result, with all three sub-roles being fairly similar, around 1/3rd of the evolutions each. This is why I dive deeper, going into detail across the information themes and issue types: averaged across enough data, equal distributions tend to arise. All results are presented in Table 5.3.

Requirements issues are more collaborative among Non-Owners. Specifically, Requirements

Content evolutions are authored by Non-Owners 31% of the time, compared to Development at 13% and Maintenance at 19%. This shows a consistent type of community involvement among requirements in ITSs that is less so in the traditional issue types Development and Maintenance. Improvement Suggestions are consistently the issue type with the most Non-Owner evolutions *across all information themes*. This suggests that this issue type is popular among Non-Owners. This contrasts with Development issues, where Owners are the majority contributors across the information themes. Specifically, Development Content evolutions are made by Owners 87% of the time, split 49/51 for Meta Content, 60% for RepoStructure, 70% for Workflow, and slightly less at 43% for Community.

Looking at the bottom three data-filled rows of Table 5.3, Epics consistently stand out as being evolved by the Creator who is also the Reporter, but *not* the Assignee (across information themes). This is particularly the case for Workflow and Community information themes. This is likely some kind of product owner or manager who is working on (evolving) the issue, despite having assigned it to someone else.

Inspecting the first scope (S1), the evolution distribution between Owners and Others is almost equal at 48% and 52%, respectively. The Owner subsets share an almost equal split as well at 34%, 37%, and 28%. These sets (Cra, CrA, cRa, and cRA) represent evolutions made by someone who is either the Creator, or the Reporter, but not both (logical XOR). This means that within the Jira Repos, considering evolutions made by a Creator or Reporter, 99.34% of the time they also hold the other role—in other words, they are "Creator-Reporters". CRA represents evolutions made by someone holding all three roles, which accounts for 34% of all Owner evolutions. CRA and crA represent all Assignee evolutions (Creator-Reporters or not), which accounts for 62% of all Owner evolutions, whereas CRa represents non-Assignee evolutions (who are Creator-Reporters), which only accounts for 37% of the Owner evolutions. In other words, Assignees make 62% of the Owner evolutions. CRa and CRA represent all Creator-Reporter evolutions (Assignees or not), which accounts for 71% of all Owner evolutions, whereas crA represents non-Creator-Reporters, which only accounts for 28% of the Owner evolutions. In other words, Creator-Reporters make 71% of the Owner evolutions.

Inspecting S2, Requirements evolutions have the highest contribution rate from Others at 57%, compared to Maintenance at 53% and Development at 42%. Looking at the Owner evolutions sets of S2 compared to S1, there are small differences. Evolutions made by CRA are lower in Maintenance issues (27%), and higher in Development issues (45%).

Inspecting S3, Content evolutions are largely made by Owners (79%), with a majority made by non-Assignees (64%), and very few made by non-Creator-Reporters (7%). MetaContent, RepoStructure, and Community evolutions are made by Others ~60%, and Workflow evolutions are leaning towards Owners at 56%.

Inspecting S4, we see Requirements Content evolutions are largely made by Owners at 69%, and even more so for Maintenance at 81% and Development at 87%. Similar to all Content evolutions, Requirements Content evolutions made by Owners are made more by non-Assignees

(59%) and very rarely by non-Creator-Reporters (7%). This pattern is more pronounced in Maintenance Content evolutions, with non-Assignees accounting for 73% of Owner evolutions, and non-Creator-Reporters accounting for only 6%. Looking at Requirements across the other evolution types in S4, Workflow evolutions are evenly split at 50/50, while MetaContent, RepoStructure, and Community have a 2/3rds distribution of Others making the evolutions. For Development Workflow evolutions, Owners account for 71% of evolutions. Other notable Requirements Owner evolution distributions include CRA Workflow evolutions at 48%, and CRa Community evolutions at 44%.

5.3.4. Patterns of Content Evolutions

Regarding the manual analysis of textual evolutions to the Summary and Description fields, I found eight content evolution types. Table 5.4 shows the distribution of the eight types across Requirements, Development, and Maintenance, sorted by occurrences in Requirements, and magnitude highlighted with a heatmap.

T1: Rewording and Refinement. The most common type, accounting for 29.8% of content evolutions analysed, was rewording and refinement of existing information in issue descriptions. Identifiers (such as version numbers) and links are updated and words are replaced by more precise ones. In a few cases, entire descriptions were rewritten, although they appeared to be reworded for clarity and not semantic changes. It was common for additional descriptive details to be added to issue descriptions, refining the formulation to narrow the focus of the statements. Identifiers such as project IDs, product names, and version numbers were added to make the statements more precise without adding new information.

T2: Adding New (Contextual) Information. A common type was to add entirely new information to the issue, usually to clarify the context. Most often, this involved appending additional sentences or paragraphs to the end of the description, which suggests that this information was discovered or requested after starting the work on the issue. Examples include adding new supporting information such as stack traces, error logs, code snippets, screenshots, and rationale for the issue or its criticality. Additionally, I observed examples of quality assurance details being added such as acceptance criteria, testing details, and steps to reproduce.

T3: Style Evolution. I observed multiple changes targeting the visual cleanliness of the descriptions. These types include removing newlines and other symbols such as dashes and slashes, adding Jira formatting such as “code” and “noformat” blocks as well as formatting hyperlinks, and converting prose sentences into bullet or numbered lists.

T4: Adding Missing Description. I observed that sometimes the description is kept empty during the issue creation and is added in a follow-up evolution. These descriptions were added anywhere from minutes to days later. In the sample, I observed that it was mostly the original issue creator who came back and added the missing description.

T5: Removing Small Details. Descriptive details were occasionally removed. Examples

Table 5.4.: Occurrences of observed content evolution types.

	Requirements	Development	Maintenance	Total	%
Issues	52	52	52	156	
Evolutions	105	115	95	315	100%
T1	41	31	22	94	29.8 %
T2	23	35	24	82	26.0 %
T3	14	26	25	65	20.6 %
T4	8	10	9	27	8.6 %
T5	5	7	3	15	4.8 %
T6	7	2	5	14	4.4 %
T7	6	1	5	12	3.8 %
T8	1	3	2	6	1.9 %

include removing Jira project information and identifiers such as release versions.

T6: Adding Meta Content. I observed multiple changes where meta content is added to the description, usually as one or two sentences at the top of the description (e.g. starting with "note that ..."). This meta-content describes the issue in the larger context of the Jira Repo or organisation. For instance, product owners mention that this issue is a duplicate, summarise related updates, or mention that the issue is addressed in a different project or ITS.

T7: Correcting Spelling, Grammar, and Punctuation. Language errors occur when describing issues. I observed that these are often fixed in follow-up edits. These content evolutions included addressing spelling mistakes, missing spaces between words, and punctuation.

T8: Adding External Links. I observed that multiple changes to the descriptions included adding links to external resources for further reading and context. Examples include reference reading, links to source code, and additional project details found on mailing lists.

5.4. Related Work

Li et al. [131] performed a preliminary literature review on evolution in requirements, summarising findings across 125 primary studies. They identified four phases of requirements evolution. The final phase, evolution tracking, involved two activities: issue management and evolution measurement. Only 7% of the studies were concerned with evolution measurement. The majority of the studies focused on other aspects such as Management Process (41%), Model Evolution (36%), and Impact Analysis (10%) [131]. These articles were predominantly concerned with traditional RE practices rather than requirements in ITSs.

Heck and Zaidman [98] performed an exploratory investigation into the evolution of Feature Requests in ITSs. Their work specifically focused on Feature Request duplicates and the reasons

that lead to them. While their work highlighted the need to understand issue evolution in ITSs, our study goes beyond just Feature Requests. I quantitatively explore the actual issue evolution in ITSs across all notable fields, comparing requirements, maintenance, and development issues and identifying several differences between Feature Requests and other types of requirements.

Jayatilleke and Lai [109] conducted a systematic literature review of requirements change management and report on the causes of change, the processes to manage it, the techniques developed to work with and avoid it, and decision-making for change management. Their results describe traditional requirements artefacts and traditional change management processes. However, the authors refer to the benefits of agile methods in change management. They describe the benefit of "minimal documentation using user stories which do not require long and complex specification documents", thus "the possibility of dramatic and constant changes is reduced". Our work showed that requirement issues did, in fact, evolve: about 8 times on average and up to 25 times. As I observe rather few changes to the content, I expect that requirements might evolve outside the ITS, leading to the creation of a new issue whenever necessary.

Bug reports are a common focus for ITS research due to their importance to the software maintenance and evolution processes in open-source ITEs. Bettenburg et al. and Zimmerman et al. studied the quality of Bug Reports and, in particular, the needs of developers using those Bug Reports [27, 250]. The primary finding from their work was the extreme mismatch between the needs of developers, and the information provided by reports. In particular, they note that incomplete information was the most common problem faced by developers [250]. This prompted a wave of research focused on improving Bug Reports. Herzig et al. investigated issues labelled as Bug Reports and found that roughly 1/3rd of Bug Reports are misclassified, leading to the conclusion that data needs to better classified [102]. Research focusing on automatically classifying issues as Bug Reports then became popular in an attempt to help ITSs and clarify data [223, 249]. This research, however, is solely focused on Bug Reports.

Multiple studies have focused on the broader needs for and usefulness of ITSs to support software organisations. This includes researchers asking what information stakeholders need to know [142] and how tool support can be tightened [79, 198, 245]. Maalej et al. found that information needs were dependent on organisational role and project phase [142]. Fucci et al. discovered issues of information overload, tool limitations, and needs to understand dependencies between issues and requirements reuse [79]. The works of Bavota and Russo [19] and of Xavier et al. [243] show how ITSs are being repurposed to manage self-admitted technical debt by using special issue types. Finally, several studies have been published about Feature Request detection in ITSs [67, 152, 207] as well as detection in other issue-like sources such as app stores [116, 143]. Our work confirms the importance of ITSs for RE practice, with both benefits and drawbacks to their use focusing on *some* issue types such as Bug Reports, Feature Requests, and technical debt. However, previously published work has covered neither the full breadth of ITSs nor requirements evolution specifically.

Finally, another related area is customer relationship management, which reflects the pro-

cess of understanding users, customers, and stakeholders in general, and how they impact the evolution of requirements. Customer relationship management involves the use of artefacts, tools, and workflows to successfully initiate, maintain, and sometimes terminate customer relationships [193]. ITSs can be used as a customer relationship management tool, managing customer feature suggestions, customer incident reports, and general support tickets [119, 152]. Previous work suggested that support tickets can be used to predict and prevent the loss of customers [161, 162, 166], furthering the value of documenting customer relationship management practices. I found that 52% of requirements evolutions are performed by stakeholders outside the ownership roles, revealing the importance of these stakeholders in the process. A deeper analysis of stakeholder interactions is required to expand on the findings presented in this article in relation to customer relationship management.

5.5. Discussion

Process-Centred vs. Community-Centred Requirements. The results reveal that requirements are prevalent in ITEs, but different projects use various types of requirements differently. While overall, Improvement Suggestions and Feature Requests are the most popular types in the entire dataset, comparing the single projects shows that Stories (combined with Epics) are overwhelmingly used by a ∼quarter of all analysed projects. This reveals at least two different *styles of requirements usage* in (OSS) ITSs: a process-centred requirements usage (such as Epics and Stories which are central to agile processes), and a community-centred requirements usage (such as Feature Requests and Improvement Suggestions which are bottom-up issues by users and other stakeholders who want or should have a say on the software). Two additional results confirm this observation. First, community evolutions occur less for Epics and Stories than for the other types of requirements, also compared to Development and Maintenance issues. Second, when requirements Content (summary and description) evolves, it is only for Epics and Stories (Fig. 5.2)—implying that these two types are likely more carefully maintained by the development team, while the others rather follow the goal of "accepting or rejecting" them, a typical elicitation and analysis goal.

Researchers and tool vendors should carefully distinguish between these different types of requirements when studying and supporting RE within ITEs. It is also interesting to understand how the requirement usages are interlinked, whether story-driven projects represent a "clean" version of requirements focusing on implementation and validation while community-driven projects focus more on elicitation, negotiation, and analysis of requirements.

Operative vs. Corrective Evolutions. The manual analysis of the large number of fields used to manage issues across 13 well-known communities sheds light on what information evolves after the creation of a requirement issue in an ITS. I reduced more than 2,000 fields in the dataset into 20 fields with distinct goals and semantics. These 20 fields represent common ground for various heterogeneous projects using the highly customisable ITS Jira. Each of the

fields by themselves can be the focus of a future comparative study, to understand, for example, how Resolution information or Comments emerge in various contexts and issue types. While some fields like Priority, Status, or Assignee have attracted notable research attention over the past decade, others, such as Links, have not. The results suggest indeed that these fields bear differences, e.g. in how long they "live" or by whom they get updated. I also grouped the issue fields into five themes, which represent different meanings likely with different impacts when changed: Content, MetaContent, RepoStructure, Workflow, and Community. This also served as a needed focus for our analysis, rather than 20 fields. Our quantitative results also reveal multiple differences between the information themes. Most notably, I found differences between Content and Workflow, although also between Content and others. For instance, Content usually only evolves within a few hours from the issue creation, while Workflow and Community usually evolve within weeks and up to months for structural information (RepoStructure) such as Parent issue or IssueType itself. Additionally, the fields have clear differences between the different types of requirements, as discussed above.

Generally, Workflow field evolutions reflect the *operative* nature of ITSs: to track status and manage workflows. Content field evolutions, on the other hand, rather reflect *corrective* actions: including the rewording and refinements of issues as well as the addition of new contextual information, needed for understanding and resolving the issue. Controlling for this difference is crucial to studying and assisting issue evolution in ITEs. So far, most research on requirements' evolution focuses on the corrective nature. The goal is to reduce the ambiguity of requirements, remove inconsistency, and "develop" the requirements from brief goals or needs into detailed use cases, user stories or models that can be validated, implemented, and tested. On the other hand, traditional ITS research, originating from bug tracking and maintenance, rather focuses on the operative evolution: on how to reduce duplicates, assist issue allocation, or reduce the resolution time as much as possible. Future work should bridge these two perspectives, as both are important, and further support and understanding are needed for both of them.

5.6. Summary

In this chapter, I investigated the information types and evolution behaviour within ITSs. I applied both quantitative and qualitative techniques to a large dataset of public Jira repos [164] consisting of 1.3 million issues, 13 million evolutions, and 2,094 projects. I performed this analysis on 1.3 million issues from 13 distinct organisations across 2,094 OSS projects. The Inductive Thematic Analysis revealed five information type themes in the data: Content, MetaContent, RepoStructure, Workflow, and Community. There are also 20 codes under those five themes that further break down and specify the information types. The historical data analysis revealed both similarities in evolution behaviour and many differences. Across all activities, issues evolve a median of ~8 times in their lifetime. Across the information themes, however, Workflow accounts for ~40% of all evolutions, whereas Content accounts for almost

none. Evolution time is not that different with requirements at 27 days after their creation (median), compared to development at 11 and maintenance at 8. When looking at information themes, however, RepoStructure was the longest evolving evolution theme (10–100 days, medians), in contrast to Content evolutions which occurred, on average, within just the first 22 hours (median). These differences in time evolution time became even more exaggerated when looking at the individual fields where some fields evolve for months (and sometimes years), whereas others consistently evolve for just a few days or weeks. The results also show that Owners and Non-Owners both evolve issue information roughly the same (48% and 52%, respectively). Finally, I observed eight types of changes occurring to the issue Content, including rewording and refinement, adding new contextual information, and style edits. I also observed that information is mostly added over time to the issue description, with very little removal or semantic rewording.

This chapter revealed the types of information and evolution that occur within ITSs. These findings further highlight the diversity of information and processes that occur within ITSs. With the collective "problem investigation" findings from Part II, we now know much more about ITEs. We know the problems practitioners face when using ITEs. We have learned specific insights regarding the inner complexities of ITSs including the artefacts, activities, information, and evolution within ITSs. Overall, we are aware of the types of problems practitioners face, and the types of environments within which they operate. In the following chapter, I transition into Part III: Solution Construction. In the first chapter, I propose a solution to these problems that considers the diversity, depth, and complexity of the studied environments. The following chapters of Part III elaborate on and further contribute to this primary solution.

Part III.

Solution Construction

Chapter 6.

Best Practice Ontology for Issue Tracking Ecosystems

> Taxonomy is described sometimes as a science and sometimes as an art, but really it's a battleground.
>
> Bill Bryson

ITS problems, complexities, and proposed solutions are all multidimensional and context-dependent (see Chapters 3, 4, & 5), yet the description of the proposed solutions hardly ever captures these dimensions or context factors. This may lead to missed information during investigations, duplicate work by researchers and practitioners seeking to address these problems, and solutions that are too vague and broad to be adequately applied. We know from the previous chapters that ITEs are: complex, rich environments with different artefacts, activities, information, and evolution, and practitioners struggle to use them effectively. We also know from research that there are many proposed solutions to individual problems in isolation from the rest of the ITE. We are missing a holistic theory that attempts to capture these complexities for structuring existing knowledge of ITSs, and for guiding future research in comparison to what exists and how to structure their findings and ITS solutions.

In this chapter, I construct an ontology that captures the multidimensional and context-dependent nature of solutions to problems in ITEs: the Best Practice Ontology for ITEs. I built this ontology using existing research on quality factors for ITEs, existing theory involving ITE "smells" and "Best Practices", and with inspiration from existing constructs in the area of quality factors for ITEs. The ontology has five sections and 16 dimensions, each of which is explained in detail. This chapter introduces and explains the ontology, and the next chapter lists a catalogue of Best Practices as collected and analysed as part of the ontology-building process. As a result of this work, future research now has a structured ontology to guide their analyses. Additionally, the ontology itself (as well as the follow-up catalogue) is now open to be challenged, falsified, and improved with future evidence and competing ontological structures. Following this chapter, I present a catalogue of ITE Best Practices, followed by listing a number of detection and repair methods for ITE Best Practices.

6.1. Research Methodology

My primary objective with this chapter is to introduce an ontological model designed to describe ITE Best Practices. For this qualitative model-building activity, I defined no research questions. For the development of the ontology, I followed the recommendations of Nickerson et al. [167] (later updated by Kundisch et al. [125]). I have described these recommendations in full in Section 2.4.2, and will only be describing here, in this section, the choices made within the framing of those recommendations. Nickerson et al. recommend seven primary taxonomy development phases, with two potential paths that can be taken depending on the "approach" chosen [167]. I selected the inductive approach (as described in Section 6.1.3), and so the seven phases I followed are: determine meta-characteristic, determine ending conditions, pick the approach, identify subset of objects, identify common characteristics and group objects, group characteristics into dimensions to create taxonomy, and review ending conditions [167]. Finally, I tested the external robustness of the final ontology by applying it to structure Best Practices that *could benefit ITEs*, but do not yet have the right framing to do so.

6.1.1. Determine Meta-Characteristic

The purpose of the meta-characteristic is to consider and define the important perspectives that your taxonomy/ontology will be a part of. This helps the construction of the dimensions such that the output is useful to the right user groups [167]. The primary reason for constructing this ontology is to support practitioners in improving the quality of their ITE, by giving them guidance in the form of information, recommended processes, and algorithms. This highlights "ITSs" as the main *artefact* of interest, "practitioners" as the main *user group* of interest, and "guiding" and "documenting" as the main *uses* of interest. A secondary reason for constructing this ontology is to support researchers in investigating and documenting how practitioners interact with ITEs, with a focus on prescribing processes that will help improve the quality of their ITE. This highlights "researchers" as an additional user group of interest, and "investigating", "documenting", and "prescribing" as additional uses of interest.

The artefacts of importance for the meta-characteristic are the *ITS overall*, and the *issues* within them. ITSs are a tool used to manage SE processes, and they are also used as documentation. ITSs are complex tools that support many use cases across various SE processes, for many stakeholder groups. As such, it is important that the ontology can handle the different use cases possible within ITSs, as well as the various processes and stakeholder groups. If the ontology is not specific enough, it won't be clear which use cases to do with which processes are affected, and which stakeholders are involved. The issues within an ITS are unique and atomic artefacts, each representing a unit of work that needs to be completed, each with properties (dimensions) that characterise it. The ontology should be aware of this complex artefact, and also utilise the structure where possible to simplify instructions and definitions.

The two primary users of this ontology will be SE practitioners and SE researchers. For

the practitioners, I have identified three different user groups: *managers* (e.g., team leads, and product owners), ITS *maintainers*, and *developers*. The manager user group is defined as someone who needs, wants, or could benefit from an overview of the quality of an ITS. The manager also adds issues to the ITS in the form of top-down plans and instructions for developers, and uses the ITS to keep track of the progress of the issues their developers are working on. The ITS maintainer user group is defined as someone who configures the ITS. The developer user group is defined as someone who utilises the ITS as a tool for knowing what needs to be done, tracking and completing tasks, and communicating with other stakeholders regarding the issues. Depending on the company context, either the manager or maintainer will also document their ITE Best Practices for (re-)use by colleagues. For the researchers, I have identified two user groups: *consumers* and *contributors*. It is also worth mentioning that "developer" is being used here to describe a broad set of roles in traditional SE, such as programmer, tester, documenter, and compliance checker. All these roles share the same user group of "Developer" since they "develop" something in response to what is described in ITS issues. The consumers user group is defined as someone who is interested in the state-of-the-art research on ITE Best Practices. The contributors user group is defined as someone who is interested in modifying or creating Best Practices based on empirical research. These five user groups outline the relevant stakeholders to be considered for the meta-characteristic.

From the five user groups above, I have identified four primary uses for the ontology: guiding, investigating, documenting, and prescribing. The ontology should support managers, maintainers, and developers by guiding them through what an ITE Best Practice is, why it needs to exist in their specific context, how to implement it, and what algorithmic support they can expect from the system. The ontology should support managers and maintainers by helping them document their existing ITE Best Practices for (re-)use by their colleagues. The ontology should support researchers in investigating existing ITE Best Practices phenomena, and subsequently modifying existing ontological items, or adding new ones. The ontology should support researchers in prescribing future ITE Best Practice behaviour, given the empirical evidence from investigating ITE usage highlights a strong recommendation for practitioners.

In summary, the *meta-characteristic* is "the information required to guide practitioners in using and documenting ITE Best Practices within their specific organisational context, and the information required to support researchers in investigating and documenting ITE Best Practices, including the prescription of future ITE Best Practices."

6.1.2. Determine Ending Conditions

I apply the recommended ending conditions by Nickerson et al. [167], both objective and subjective. There are eight recommended objective ending conditions and five recommended subjective ending conditions. See Section 2.4.2 for a full list of these ending conditions.

6.1.3. Pick the Approach

I chose the inductive approach as the primary means of creating the ontology, but I also integrate deductive thinking into the overall design. The concept of Best Practices for ITEs is not new (although not yet framed this way). The concept of "smells" and "recommendations" for ITEs are certainly not new [27, 187, 188, 250]. Considering this existing research, I decided that the best approach to building this ontology was to review existing practices for discussing ITE quality, and combine that knowledge into a single ontological model. However, as I have been studying this topic for my entire PhD, I also have a significant amount of knowledge and understanding of both ITEs and the state-of-the-art research. Thus, I decided to integrate deductive passes over the ontology during the overall inductive approach. My decision when to incorporate these deductive passes was not systematic or based on clear criteria, but as an additional component not required by the inductive approach, I viewed these passes as strictly beneficial with no clear downside. I would compare my deductive passes to that of Phase 4 of Thematic Analysis (as proposed by Braun and Clarke [33]), whereby as the researcher you need to step back, and ask yourself if the model that is forming is cohesive and representative of your goal. Without such a perspective, it is possible for your inductive analysis to produce something empirically grounded, but missing higher-level meaning and purpose [33].

6.1.4. Identify Subset of Objects

Ontology development can be either inductive or deductive (see Section 2.4.2). I chose the inductive approach because there is an abundance of empirical data on ITEs and the quality thereof. I defined which types of objects are important for the taxonomy, and which subset of the population to sample. For the objects of interest, I identified two types of objects that are important for the taxonomy: existing classification systems for quality aspects of ITE and the quality aspects themselves. The **first** type represents the quintessential objects that this ontology is designed to unify and improve. These are existing efforts to structure quality aspects of ITEs as units (e.g., communication and analysis). My ontology is designed to capture, unify, and structure this area of research. The **second** type represents the unstructured form of the objects that this ontology seeks to organise. These objects can (presumably) be structured to fit into the taxonomy easily, and act as *internal robustness checks* during the ontology creation process (thereby impacting the formation of the ontology). Each paragraph below describes these types in more detail, and describes the subset of the population that was sampled.

The first object type contained articles structuring and specifying ITE quality aspects, and served as the primary ontology development dataset. The goal for this type was to review existing models for structuring and reporting best-practice-like concepts, which could then inform the creation of the ontology. To find these objects, I searched for articles that discussed some named concept synonymous with "ITE Best Practices" such as "recommendations", "smells", "patterns", and "antipatterns". While there is an abundance of literature involving quality as-

pects for ITEs, it is rare that these articles also attempt to structure the objects beyond giving each a name and a description. Therefore, I based the data collection for this rare object type on key literature and snowballing. I started with key literature on the topic—well-known and recent literature on ITS Best Practices, and then I snowballed forward and backwards searching for similar research. This approach is acknowledged by Nickerson et al. who describe the collected objects for ontology development "likely to be the ones with which the researcher is most familiar or that are most easily accessible [...] a convenience sample" [167]. The goal for collecting this type of object, however, was not completeness. While collecting as many articles as I could find was the goal, I did not follow a strict secondary-study protocol [123, 179, 180] that would have increased the confidence and validity of a completeness claim. Despite the lack of knowledge regarding completeness, the first object-type dataset was collected using acknowledged rigorous methods, and is sufficient for the purpose of forming an ontology [167].

The second object type contained quality aspects for ITEs, regardless of the structure used to discuss them. The goal for this type was to iteratively check the robustness of the ontology *during development*. In other words, to check if the ontological properties could handle the important parts of each object, and that the properties were differentiating the objects well enough [167]. This object type was intended to only include "internal" objects, which are objects that are already within the context of quality aspects for ITEs. Given the shared context, these objects should be easily structured and specified by the ontology. To find these objects, I started with the articles from the first object type, which each already had such quality aspects.[1] I collected additional articles that described ITE quality aspects without mentioning a structure beyond a name and a description. The data collection process for these articles was also based on key literature. The study of ITSs, and the broader environment and context of ITEs, is a well-studied area that has been popular for the past 20+ years [27, 250]. Similar to the first object type, the goal here was not completeness. Rather, the goal with this object type was to check the internal robustness of the ontology.

6.1.5. Identify Common Characteristics and Group Objects

Some common characteristics arose from the articles discussing quality factors for ITE Best Practices. First, it was common for the articles to discuss something negative that is to be avoided. This is commonly referred to as a "smell" across different areas, including ITS quality and code quality (among others). However, even without a mentioned construct, *avoiding negative outcomes* appeared to be the main driver behind many articles describing quality aspects for ITEs. Second, it was common for the articles to discuss some kind of *algorithmic contribution*. It seems most SE researchers are interested in engineering solutions to the quality issues that exist in ITEs. Combined, these two concepts (avoiding negative outcomes and

[1] I did not find any articles for the first object type that only discussed structure, without also noting examples of ITE quality aspects from previous articles or ones from new empirical work conducted in the article.

algorithmic detection) were by far the most common characteristics to be discussed.

Less common, but still present in some articles, is the concept of recommending some kind of process to avoid these negative outcomes. This could be as simple as recommending the implementation of the algorithmic detection, or as complicated as recommending a process change within the company, or a configuration change within the ITS itself. The classic "What Makes a Good Bug Report" by Zimmerman et al. [250] recommends both process changes (such as "provide feedback on Bug Reports") and ITS configuration changes (such as "provide tool support to collect information"). Finally, present in some articles, but exceedingly rare, was the characteristic of context. Given the meta-characteristic described above in Section 6.1.1, and my findings from Chapters 3, 4, and 5, I was looking for *contextual factors* to focus the discussion on a particular set of organisational factors. Most articles do not discuss context at all, which implies that their ITS quality factor discussion applies to all ITSs in all organisations. Those rare articles that did discuss context, were largely reported in the methodology section when describing the organisation, ITE, or dataset they were working with. There was minimal discussion of context and how it applies to the findings and recommendations when it comes to quality factors in ITEs. Despite the rarity of this characteristic, some articles did address context, including Eloranta et al. [53], who explicitly model context as a dimension in their catalogue of scrum antipatterns. Overall, I noticed the concepts of avoiding negative outcomes, recommendations to be better, and contextual factors. These characteristics are well aligned with the meta-characteristic, supporting the activities of guiding and prescribing.

6.1.6. Group Characteristics into Dimensions to Create Taxonomy

The results of this phase in the iterative process is the ontology itself. Every iteration led to the creation and refinement of dimensions based on the meta-characteristic described in Section 6.1.1 and the characteristics that began to form in Section 6.1.5. For full details of the formed dimensions of the final ontology, see Section 6.2 below.

6.1.7. Review Ending Conditions

Creating this ontology was an iterative process that took many cycles, each time reviewing the ending conditions listed in Section 2.4.2. As the subset of reviewed objects is not the full population, it was also possible to find more objects and continue the cycles. However, for the purpose of creating this ontology, the initial sample of objects was sufficient for creating a thoroughly grounded ontology.

6.2. The Ontology

The ontology has five sections and 16 dimensions. The five sections are Meta, Summary, Recommendation, Context, and Violation. I structure and summarise these sections and dimensions

in Table 6.1. In the following subsections, I describe each ontology section and dimension, with a focus on linking them to the meta-characteristics when possible.

Ontologies can have a mix of properties (dimensions) and relationships. As described in Section 2.4.2, the most important focus when building an ontology is the specific purpose for which it is built. In the ontology presented in this chapter, the primary purpose is to structure and convey specific types of information, which an ontology does well through properties. The only relationships conveyed through this ontology are the "broader/narrower" relationships naturally described by the taxonomy-like structuring that exists in the grouping of the objects.

6.2.1. Meta

The Meta section describes the ontological structure of the Best Practice itself. The dimensions within the Meta section are the *name* and *source* of the Best Practice. The name is a succinct and distinct natural-language way to refer to this Best Practice. The Best Practice name should be both catchy and intuitive, such that the whole Best Practice is well summarised within the name itself. The source is the place where this Best Practice comes from. This is often a research article, but it can also be a blog post or an organisational statement. The source dimension structures and encourages backwards traceability to where the Best Practice began. Importantly, this dimension is not limited to a single value, and should instead list all sources that have contributed to the Best Practice as it is currently formed.

6.2.2. Summary

The Summary section describes the high-level purpose of the Best Practice. The dimensions within the Summary section are the *objective* and *motivation* of the Best Practice. The objective is a succinct description of the ultimate goal of applying this Best Practice. For example, "Achieve good Bug Report quality" is a common objective found in nine of the Best Practices in the catalogue.[2] The motivation is a longer explanation of the concepts involved with this Best Practice, including the background, gaps, problems, solutions, where it came from, and potential benefits. The purpose of this field is not to be complete in reporting on the Best Practice, but rather to justify (to "motivate") the need for such a Best Practice. While the need for a Best Practice *might* be clear from the objective, it is not always obvious why such an objective is necessary or desirable. The motivation is designed to be a dimension open to a little interpretation that has flexibility built in to how it is written for a given Best Practice.

6.2.3. Recommendation

The Recommendation section describes the actions that should be taken to follow the Best Practices. This section is catering directly to the "guiding" and "prescribing" aspects of the

[2]The abundance of Best Practices with this objective can be explained by the SE research community's focus on Bug Report quality for the past 20 years [27, 94, 250].

Table 6.1.: Best Practice Ontology for ITEs.

Meta*
Name: The name of the Best Practice.
Source: Where the Best Practice came from; often a research article.
Summary
Objective: The purpose of the Best Practice; the goal of implementing it. If possible, don't describe what needs to be done, but rather what the (positive) outcome will be.
Motivation: Explanation of the concepts involved, including the background, importance, and potential benefits.
Recommendation
Process: The recommended steps to take to follow the Best Practice.
ITS: Specific instructions or configurations for the ITS involved.
Context
Stakeholder Benefits: Outline who benefits from this Best Practice being implemented. Separate each stakeholder group who is affected differently. Be clear by first naming the stakeholder group, followed by the benefits.
Stakeholder Costs: Same as "Stakeholder Benefits", except, outline the costs associated with different stakeholder groups.
ITS Scope: The scope of the ITS required to understand the conformance to this Best Practice. Examples include a single issue, pairs of issues, and the entire ITS.
Issue Types: The specific issue types this Best Practice applies to. Examples include Bug Report, User Story, Epic, and Work Item.
Inclusion Factors: Specific context factors that *should* be fulfilled for this Best Practice to be a good recommendation for a specific individual, project, team, or organisation.
Exclusion Factors: Specific context factors that *should not* be fulfilled for this Best Practice to be a good recommendation for a specific individual, project, team, or organisation. These factors can be explicit exceptions to the Inclusion Factors above, or they can be key indicators that this Best Practice is not well-suited to a given context.
Violation
Smells: The outcomes that suggest this Best Practice is being violated. These are not necessarily *negative* outcomes, but good first indicators that something is wrong.
Consequences: Potential negative outcomes that could happen as a result of not following this Best Practice.
Causes: Potential reasons why the Best Practice was not followed, leading to a violation.
Algorithmic Detection: The pseudocode that automatically detects violations of the Best Practice in the associated ITS. If possible, actual code is also appreciated.

* In the catalogue presented in Chapter 7, the Meta information (name and source) are listed as a title above the table, and are not included in the table itself.

meta-characteristic for practitioners. The dimensions within the Recommendation section are the *process* and the *ITS recommendations* of the Best Practice. The *process* is a simple description of the steps an individual, team, or organisation would take to follow this Best Practice. For example, "BP24: Link Duplicates" (see Table A.25) has a recommended process of "always include a link to the duplicate bug when referencing it in another bug." It should be short and to the point, and exclude the complexities of implementing it in teams and organisations. The *ITS recommendation* includes specific instructions or configurations for the ITS involved. For example, the "Link Duplicates" Best Practice mentioned above has the ITS recommendation of "have a separate 'duplicated by' like type and use that when referencing duplicate bugs". The reason the ITS recommendation is separate from the process recommendation is due to the artefact of the meta-characteristic: the ITS. The ITS is a central, dynamic, and configurable tool at the heart of organisations implementing these ITE Best Practices, and should be leveraged. Recommendations for the ITS are then central to the conceptualisation of ITE Best Practices, and in most cases there should be an accompanying recommendation for the ITS.

6.2.4. Context

The Context section describes important factors that affect the application of the Best Practices. With just the Summary, Recommendation, and Violation sections, it is possible to understand and apply the Best Practice principles. However, these outcomes would be the result of many hidden factors at play, which happen to be in place for the Best Practice to be successful. The Context section holds the answer to the classic response to the application of Best Practices and smells in organisations: "it depends". In Chapter 3, I found that many problems discussed by the practitioners were founded on an accepted uncertainty with which to interpret their statements. These problems, and the Best Practices that may help mitigate them, are reliant on hidden context factors that can be surfaced. Using the meta-characteristic, and the iterative cycles of inductive and deductive reasoning from the ontology creation process, I surfaced six dimensions that form this Context section: Stakeholder Benefits, Stakeholder Costs, ITS Scope, Issue Types, Inclusion Factors, and Exclusion Factors.

The *Stakeholder Benefits* dimension describes who benefits from this Best Practice being implemented. As a starting place for the question "would this Best Practice be good for us?" knowing who would benefit from such a practice is key. If no stakeholders in your organisation would benefit, then the Best Practice should not be applied. Conversely, multiple benefiting stakeholder groups implies a strong reason to implement the Best Practice. This dimension should be described from the perspective of individual stakeholder groups. Separate each stakeholder group who is affected differently by this Best Practice. Be clear when describing the benefits by first explicitly listing the stakeholder group, followed by the benefits. For example, "BP01: Good Bug Report" (see Table A.2) describes the stakeholder benefits as: "*Developers*: Get the information they need. *Reporters*: Get their reports resolved faster."

Benefits for one group of stakeholders often come at the cost of other stakeholders. This is the reason for the second Context dimension: Stakeholder Costs.

The *Stakeholder Costs* dimension describes who pays a cost for the implementation of the Best Practice. This could be a cost associated with setting up the Best Practice, enforcing the Best Practice, or interacting with it daily. For example, "BP01: Good Bug Report" (see Table A.2) describes the stakeholder costs as: "*Reporters*: Have to put more effort into submitting the required information." This Best Practice is quite pervasive in the OSS community in places like GitHub, where the submission of a Bug Report usually starts with a template that you must fill out. The template requests information such as software version, computer version, expected behaviour, actual behaviour, screenshots of GUI issues, and sometimes even a minimum working example of a development project that experience the bug. This requires a lot of time to gather, which is a cost for the reporter that needs to be acknowledged. Given the pervasiveness of the "Good Bug Report" Best Practice, this cost is assumed to be reasonable. Same as the Stakeholder Benefits, the Stakeholder Costs should separate each stakeholder group who is affected differently by this Best Practice. The result of the first two Context dimensions then is two lists of stakeholders and how they benefit from or pay for the Best Practice.

The *ITS Scope* dimension outlines the scope of the ITS required to understand the conformance to the Best Practice. Given the central nature of the ITS to ITE Best Practices—as outlined by the meta-characteristic highlighting the ITS as the primary artefact of interest, it is important to understand how much of the ITS is involved with the Best Practice. For example, the "BP01: Good Bug Report" (see Table A.2) Best Practice only requires a single issue to check conformance, while "BP24: Link Duplicates" (see Table A.25) requires the entire ITS to check conformance. This dimension, then, acts as a proxy for complexity of the Best Practice, including things such as algorithmic detection complexity, understanding its impact, and applying it in practice. As a starting place for organisations looking to improve their ITE through Best Practices, it is recommended to start with Best Practices where the scope is a single issue. Once organisations become comfortable with ITE Best Practices, they can look to implement more complex ones with greater ITS scopes.

The *Issue Types* dimension outlines the specific issue types this Best Practice applies to. As fully described in the meta-characteristic, ITSs are complex systems that manage many different processes within an organisation. In Chapter 4, I found that those activities spanned the SE lifecycle, including requirements, development, and maintenance. These different activities and processes are largely delineated by the "issue type" assigned to every issue. In this way, issues with different types represent very different artefacts and activities within an ITS, and therefore should be treated differently and quite distinctly by the Best Practices. While a Best Practice can apply to all issue types, such as "BP16: Respectful Communication" (see Table A.17), many Best Practices only apply to specific issue types, such Bug Reports. Accordingly, the Issue Types dimension of the Context section captures this information, and can support organisations deciding which Best Practices to use. If an organisation only manages Bug Reports

in their ITS, then only Best Practices that apply to Bug Reports should be considered.

The *Inclusion Factors* dimension describes specific context factors that *should* be fulfilled for this Best Practice to be a good recommendation for a specific individual, project, team, or organisation. The first four Context dimensions listed above are specific factors that should be considered for the inclusion of Best Practices within a given context; however, given the diversity of ITE Best Practices and the organisations that implement them, it is appropriate to have a general catch-all dimension where additional inclusion factors can be described. For example, many of the Best Practices including "BP32: Ordered Product Backlog" (see Table A.33) and "BP34: Story Points Over Hours" (see Table A.35) only apply to organisations that apply agile practices. Accordingly, these Best Practices have some form of "Team uses Agile" or "Team uses Scrum" listed within the Inclusion Factor dimension. If an organisation does not apply Agile, then these Best Practices have no meaning to them and should not be applied. Many of the Best Practices listed in the catalogue in Chapter 7 have "None" as a value for this dimension. However, the value of "none" is likely due to a lack of empirical evidence and understanding of the Best Practices, rather than because there are no inclusion factors.

The *Exclusion Factors* dimension describes the specific context factors that *should not* be fulfilled for this Best Practice to be a good recommendation for a specific individual, project, team, or organisation. These factors can be explicit exceptions to the Inclusion Factors, or stand-alone exceptions. For readability and understandability, it is preferred to frame an arbitrary context factor in the positive and put it in the Inclusion Factors, rather than framed in the negative and included in the Exclusion Factors. The reason for both inclusion and exclusion factors is the need to negate portions of Inclusion Factors. For example, "BP37: Recommended Sprint Length" (see Table A.38) has the Inclusion Factor "Team uses Agile (Scrum)", and the Exclusion Factor "If the development must be synchronized with external work that has slower pace (e.g. hardware development), longer Sprints may be justified". So both the inclusion and the exclusion factor are needed to describe these additional context factors that affect the application of this Best Practice. Both the Inclusion Factors and the Exclusion Factors are a key component of the meta-characteristic when it comes to *prescribing* Best Practices to practitioners. Without such dimensions, the ontology is missing a critical piece of its core purpose. This can also be extended to imply that Best Practices with no listed inclusion or exclusion factors are also missing a critical piece of their core purpose.

6.2.5. Violation

The Violation section describes aspects of the negative consequences that come from not following this Best Practice. The concept of violating the Best Practice is important. While the positive outcomes of the objective described in the Summary section should be motivating enough, including the negative consequences forms a full picture of the potential effects of the Best Practice on the organisation. Additionally, as described above in Section 6.1.5, a

major motivator for the Violation section is the current "negative to be avoided" rhetoric in the research community regarding ITS quality factors. In other words, it is popular to propose "smells" that are to be avoided, instead of "practices" that are to be followed. To capture these aspects of the Best Practices, the Violation section has four dimensions: *smells*, *consequences*, *causes*, and *algorithmic detection*.

The *smells* dimension represents the classic representation of ITS quality factors as "smells", and lists the outcomes that suggest the Best Practice is being violated. The concept of "smells" in SE nomenclature is well described, understood, and utilised, as fully detailed in Section 2.3. With the placement of smells inside this ontology, it should be understood that the concept of a smell is a subset of what an ITE Best Practice describes and offers. To create and describe an ITE Best Practice, much more information is needed, and much more thought must be put into the process. For ITS quality factor recommendations (including those aimed at the broader ITE), it is possible and beneficial to frame them as an ITE Best Practice instead of a Smell. This is not based solely on the requirement of more information and thought, it also has to do with what is lacking in the concept of a Smell itself: context. I elaborate on this in Section 6.3, where I introduce a set of Propositions.

The *consequences* dimension lists the potential negative outcomes that could happen as a result of not following the Best Practice. For example, the consequences of violating the "Good Bug Report" A.2 Best Practice includes "developers are slowed down by poorly written Bug Reports". For *researchers*, this is a prime area for *investigating*: do these consequences normally occur as a result of violating this Best Practice, and are they negative?[3]

The *causes* dimension highlights the potential reasons why the Best Practice was not followed, which subsequently lead to the violation. This is an area of interest to both practitioners and researchers, as the answer to "why" questions are both interesting and useful. If an organisation is trying to follow a Best Practice, and they are following the recommended process, but not seeing the positive outcomes described in the objective, they need to know where to look for problems. The "causes" dimension is such a place where potential reasons for violations should be listed. For example, using the "Good Bug Report" A.2 Best Practice again, the causes include "reporters don't know what information to provide, reporters don't want to put in the time to provide all required elements, and reporters don't know hot to get all the required elements". Investigating these causes in an organisation will likely lead to necessary changes required to see the positive outcomes from this Best Practice.

Finally, the *algorithmic detection* dimension lists pseudocode that automatically detects violations of the Best Practice in the associated ITS. Actual code is also appreciated, but rare and perhaps less useful given that pseudocode is language-agnostic. The difference between smells and algorithmic detection is the certainty with which they operate. A smell is often described

[3]The classic example being the assumptive statement that "passive voice leads to ambiguity in requirements", which, although is a potential consequence of a Best Practice violation, is highly debated whether it actually does lead to ambiguity, and whether that ambiguity is negative or not [74].

as a high-level conceptual aspect to look out for, while algorithmic detection should describe specifics that can actually be coded. Even if a smell is specific in what it describes (which is not uncommon), the smell is still describing a situation in which the Best Practice *might* be violated. The algorithmic detection, on the other hand, should only reveal situations in which the Best Practice has indeed been violated.

6.2.6. The Ontology Modelled with Thesis Constructs

I created the ontology to unify important and related constructs, and address particular shortcomings discussed in previous chapters. Figure 6.1 models these constructs alongside the ontology using a UML class diagram. This model visualises how the ontology is related to these constructs. Figure 6.1 begins by modelling the three primary constructs: Issue Tracking Ecosystems (ITEs), Issue Tracking Quality, and the Best Practice Ontology.

ITEs are aggregates of *stakeholders*, *processes*, and an *ITS*. The Stakeholders include the SE organisation, developers, managers, and clients. The Processes include all workflows that govern how the stakeholders interact with the ITS. The ITS is composed of *issues*, each of which has an *issue type*. Together, these form the basic conceptualisation of the primary systems under study. The full extent of ITSs constructs is discussed in Chapter 2.

Issue tracking quality is described by different *quality factors*, for example, *Traceability of Decisions and Outcomes* and the *Findability of Information*. All the listed quality factors can be found in the catalogue of Best Practices for ITEs in the next chapter, including many more quality factors not listed in Figure 6.1. Issue Tracking Quality can be obtained by following well-defined *Patterns*, which themselves include a description of *context*, *problem*, and *solution*. The Context section of a pattern describes the expected factors that need to be present before the problem is likely to occur. The Problem section describes an undesirable situation, such as a specific type of poor quality output. The Solution section describes how to fix or improve the situation once the problem has been found. *Smells* signal a potential problem, but are lacking an awareness of context, which adds uncertainty to the detection. Together, these concepts form the basic conceptualisation of Issue Tracking Quality in this thesis.

The **Best Practice Ontology** itself is described above in Section 6.2. This ontology relates to ITEs and Issue Tracking Quality in multiple ways. The *Objective* describes the desired issue tracking quality. The *Recommendation* section models the solution part of a pattern. This includes the process recommendations for ITE processes (to adhere to the Best Practice), and the ITS configuration recommendations. The Context section models the context part of a pattern. This includes benefits and costs for ITE stakeholders, the ITS scope this Best Practice applies to, and the relevant issue types. Finally, the Violation section models the problem part of a pattern. This includes the Smells which models the concept of a smell, and the Algorithmic Detection which describes how to automate the detection of violations within the ITS.

Overall, the Best Practice Ontology for ITEs is highly coupled with the core concepts presen-

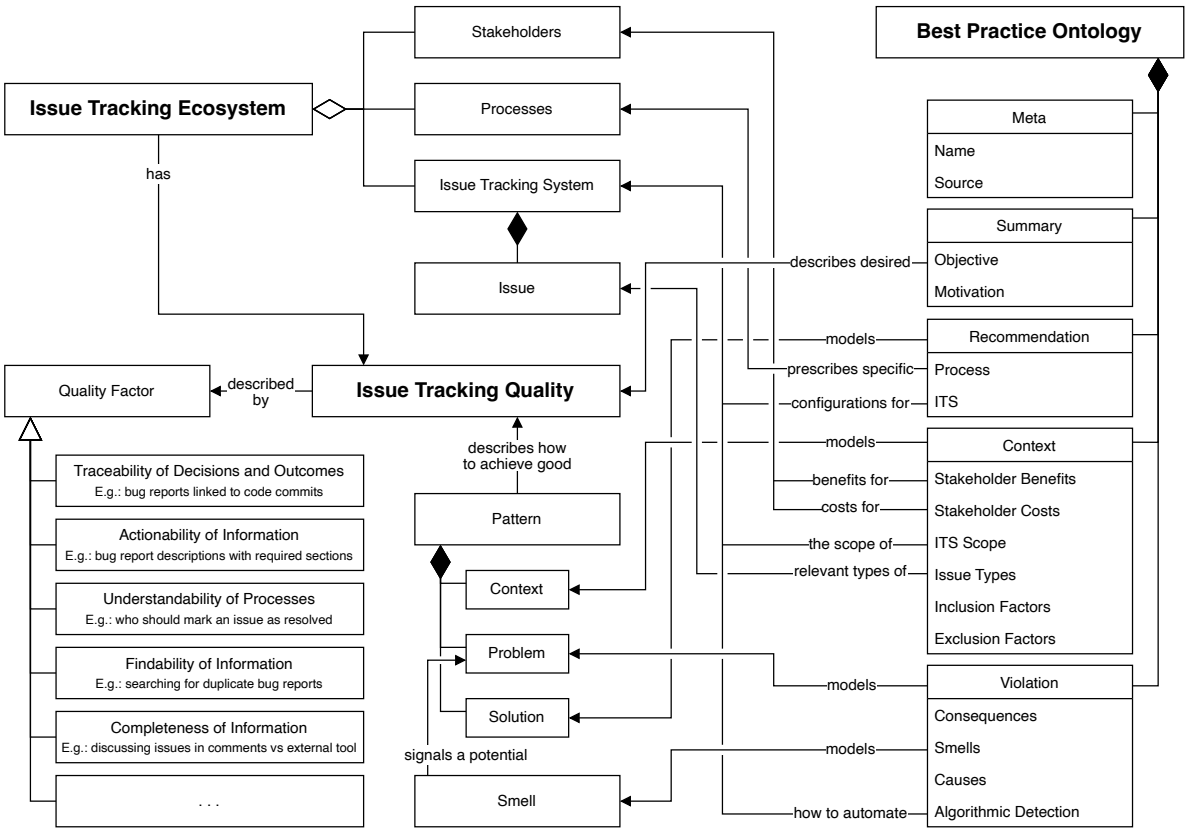


Figure 6.1.: Conceptual model of ontology and thesis constructs.

ted and investigated in this thesis. It draws inspiration from—and models—existing constructs such as Patterns and Smells. It captures and describes core concepts such as the stakeholders and processes of ITEs. Most importantly, the ontology models these concepts in discrete categories, which offers a unified way to research, discuss, and apply these Best Practices.

6.3. Propositions Towards Future Theory

In this section, I discuss the Best Practice Ontology for ITE in relation to ontological theory, and what claims I seek to make with such a construct. In Section 2.4.2, I discuss taxonomic theory (which I refer to here as ontological theory), and in Section 2.4.2, I introduced the concept of propositions in theories. I combine all these concepts together, to form a set of propositions towards future theory in the area of ITE Best Practices.

The line between an ontology, and ontological theory, is not well described in the literature [189]. In Section 2.4.2, I discuss taxonomic theory, including the importance of grounding taxonomic and ontological models in rigorous secondary studies. Ralph states that not all taxonomies are taxonomic theory [189], but himself does not draw a clear line—as perhaps one is not possible, and requires consideration in each individual case. As such, I will make my own attempt at drawing the line for this specific ontology. I believe that this ontology lacks the rigorous data collection necessary to claim this ontology is also a theory, from the perspective of the data used to develop it. There are other conditions for which an ontological theory should be evaluated, but the data from which it was formed is a central place of scrutiny [189]. Another place of scrutiny, as discussed by Sjøberg et al. [211] and Runeson et al. [195], is "empirical support": "the degree to which a theory is supported by empirical studies that confirm its validity". While the formation of the ontology has been conducted using known rigorous secondary-study methods, there have been no follow-up studies on the theory itself to provide any empirical data on its utility or effectiveness. For this reason, I refrain from calling this ontology a theory. Instead, I frame five propositions, working towards a future theory.

I do not claim that these propositions form a theory, nor do I claim that these propositions are all a subset of a single theory. Rather, my claim is that these propositions each hold on their own, and some may be combined into a larger theory one day. With additional work, empirical results, and reflection, these propositions can be the start of an ontological theory of ITE Best Practices. More importantly, however, these propositions should be challenged: refuted, strengthened, or replaced. In the process of doing these things, our knowledge about quality factors for ITEs will grow and mature.

Proposition 1. The Context dimensions enable falsifiability.

It is easy to claim, frame, and create a "Smell" because all you need is a single context in which it *might be perceived* to be a problem, and the Smell holds. This has many benefits, including the quick definition and recommendation of Smells to practitioners. However, from a scientific perspective, Smells lack falsifiability, and therefore lack the ability to be challenged and improved over time. It is possible to continually add potential contexts in which people believe a smell could apply. However, Smells are—by definition—suppose to be indicators of potential problems, to be investigated by the stakeholders within their own given context, who then act as decision makers regarding how problematic the situation is, and how to act on that information. That is both the benefit and the downside of Smells. ITE Best Practices, on the other hand, explicitly model context across multiple dimensions, with the strong statement that a violation *is a problem*, and it holds under the specific context factors described. This has downsides as well, including the difficulty involved with identifying and empirically validating these contextual factors. However, that is precisely the purpose of empirical falsifiability in science: evidence needs to be gathered that supports or refutes the claims (propositions), such that over time the claims are strengthened or weakened. I believe it would be a mistake to apply this perspective to Smells by extending their definition or application, and instead I believe ITE Best Practices serve this purpose well.

Proposition 2. The Context dimensions positively affect Best Practice adoption.

One of the primary uses for the Best Practice Ontology is to structure Best Practices that will be used in industry. The Context dimensions explicitly model factors that need to be considered when implementing the Best Practice (or whether to implement them at all). While the openness of Smells is a benefit that it doesn't restrict too much, it also does not provide guidance regarding where and when to implement them. ITE Best Practices, on the other hand, give plenty of implementation guidance in the form of Stakeholders, ITS issue types, ITS scope, and assorted inclusion and exclusion factors. A well-formed ITE Best Practice should sufficiently answer the practitioner question: "Should we implement this in our context?". Therefore, I posit that the Context dimensions positively affects Best Practice adoption.

Proposition 3. The positive framing of the Summary and Recommendation sections positively affects Best Practice adoption.

Both the Summary and the Recommendation sections are designed to explain what the Best Practice does and how to follow the practices, from a positive perspective. In other words, they recommend what the goal is, and how to achieve it. This is in contrast to Smells, which describe what problem could exist, and how to avoid it. For most trivial recommendations, they can be equally worded in the positive or the negative. For example, the "BP01: Good

Bug Report" (see Table A.2) Best Practice could be just as easily described as "avoid long paragraphs without structure, and don't miss important information such as software version number", and called a "Messy Bug Report" Smell. Over time, however, an organisation selecting many things *to avoid*, does not form the same goal-oriented mindset as selecting many things *to aim for*. While framing the negative things to avoid helps those trying to avoid them, it doesn't help guide people. As described above, it will take more effort to understand where you want to go as an organisation, and therefore which Best Practices to select and curate; however, the result is a much clearer understanding of organisational goals. Additionally, achieving goals is akin to positive reinforcement, which "enhances employee performance by encouraging desired [behaviours] and eliminating negative ones" [239]. Avoiding a negative outcome is akin to not breaking rules, or abiding the law, which is generally expected of everyone and not something to be rewarded. Therefore, I posit that the positive framing of the Summary and Recommendation sections positively affects Best Practice adoption.

In the ITE Best Practice catalogue, there are some Best Practices that are worded in the negative, and some that are even worded in both forms. For example, "BP14: Active Bug Reports" (see Table A.15) compared to "BP13: Avoid Zombie Bugs" (see Table A.14), which are the same underlying Best Practice, just worded in the positive and in the negative. Admittedly, this is not the purpose of ITE Best Practices; however, this initial catalogue is designed to showcase the structuring ability of the ontology, not showcase perfect Best Practices. With so many recommendations worded negatively, I decided to include a few negatively framed Best Practices that were rather difficult to frame positively. Additionally, I included the above example with both a positive and negative framing because I wanted to showcase an interesting case that deserves further analysis. With that in mind, I hope these Best Practices are quickly challenged, updated, and added to, as per the primary function of the ontological structure.

Proposition 4. The Stakeholder Benefits dimension positively affects Best Practice compliance and satisfaction doing so.

SE is ultimately about building tools for users, which hopefully support the users and make their life easier. Research has shown that developer's awareness of user involvement and feedback positively affects their satisfaction [6, 17, 252]. Research has also shown that employees gain job satisfaction when they prioritise collective interests [49], such as helping each other within the organisation. With these perspectives in mind, it can be deduced that knowing who is positively impacted by a Best Practice will increase both the likelihood that the employee follows the Best Practice, and that they are satisfied doing so. This is in contrast to Smells, where following the recommendation will benefit "management", in some broad, disconnected sense. Therefore, I posit that the Stakeholder Benefits dimension positively affects Best Practice compliance and satisfaction doing so.

Proposition 5. The Stakeholder Costs dimension increases empathy when considering the implementation of a Best Practice.

ITE Best Practices explicitly model who will be negatively impacted by the implementation of the Best Practices. Research has shown that knowing who is impacted can enhance empathic responses due to the "neural activation shared between self- and other-related experiences" [23, 210]. In this way, by revealing whom to account for when considering the implementation of a Best Practice, the Stakeholder Costs dimension will likely increase empathy and therefore understanding of the tradeoffs involved. Without such a mechanism, it can be tempting to put too many restrictions or Smell detections, with the hope of improving the quality of ITE, without knowing—or caring—who is negatively impacted. Therefore, I posit that the Stakeholder Costs dimension increases empathy when considering the implementation of a Best Practice.

6.4. Summary

In this chapter, I formed the Best Practice Ontology for ITEs using an approach that utilises both inductive and deductive cycles. The resulting ontology can be used to structure existing research on quality factors for ITSs, and support future research and usage of Best Practices for ITEs. The need for this ontology was investigated in previous chapters. The Context section in particular is the direct result of findings from Part II, where the results consistently and repeatedly revealed the prevalence of diversity and complexity in these systems. This includes the diversity and complexity of the problems presented by practitioners using ITSs, the artefacts and activities in ITSs, and the information and evolution within ITSs. When we showed the results of these Best Practices to practitioners, they agreed with the existence of most of the smells, and further confirmed the context-sensitive nature of when these smells occur and when they are problematic. With the ontology now formed, the next step is to apply this ontology to existing literature on quality factors for ITEs. While the ontology is a useful tool for researchers, practitioners will get much more value out of the formed Best Practices themselves. In the next chapter, I created a catalogue of Best Practices for ITEs.

Chapter 7.

Catalogue of Best Practices for Issue Tracking Ecosystems

> Research is seeing what everybody else has seen, and thinking what nobody else has thought.
>
> Albert Szent-Györgyi

In this chapter I collect, structure, and present a catalogue of 40 Issue Tracking Ecosystem (ITE) Best Practices. In the previous chapter (Chapter 6), I proposed the Best Practices Ontology for ITE, but not any actualised Best Practices. Previous research has contributed many solutions and insights to improve the state of ITS and their ecosystems, but they do not structure them in such a way that my ontology does. To highlight one dimension of value my ontology brings, and to contribute a starting place for the actualised ontology, I present here a catalogue of ITE Best Practices. I leverage existing research on ITS "smells" and "Best Practices". I investigate this area to collect relevant findings that are lacking a comparable structure. The results of this catalogue building process are 40 ITE Best Practices. The findings of this analysis contribute a concrete list of ITE Best Practices, which the community can now utilise, build on, challenge, and update over time. Following this chapter, I present 18 algorithms designed to automatically detect and repair violations of the ITE Best Practices presented in this chapter. While specific Best Practices within this catalogue already recommend algorithmic approaches that should be followed to detect violations, these recommendations are often pseudocode or just plain-text descriptions.

7.1. Research Methodology

My primary objectives with this chapter are to *form* and *validate* a catalogue of ITE Best Practices that conform to the ontology I introduced in Chapter 6. For the artefact-based catalogue formation process, I do not propose any research questions. The goal is to produce catalogue items, and by the nature of artefact-based research, it will be successful. For the validation of the catalogue, I formulate a single research question:

RQ1 How do practitioners perceive issue *smells*: do they occur and are they problematic?

I explain the two methodology stages in greater detail in this section. For the formation of the catalogue, I describe the dataset used to create the catalogue, and I outline four methodological considerations for the final catalogue. For the validation of the catalogue, I explain the interviews I conducted, including which constructs I asked them about. Notably, I asked them about *smells* and not *violations* or *Best Practices*.

7.1.1. Forming the Catalogue: Inductive Extraction

Methodologically, I chose to construct the catalogue using an inductive approach, utilising existing research in the domain of ITE (compared to a deductive approach, constructing the catalogue from high-level principles, common knowledge, or personal experience).

Catalogue Dataset

I used two primary datasets to extract the Best Practices: 1) the set of articles from Chapter 6 that also contain itemised Best Practices, and 2) articles that do not offer any ontology-like structural elements but still list itemised Best Practices.

The first set of articles from which I extracted Best Practices is the same set from Chapter 6, the formation of the ontology. These articles were selected for the ontology creation because of the structural elements they impose on Best Practices; furthermore, they also contained Best Practices themselves. I list them here, in chronological order of publication date:

2013 Heck and Zaidman: "An Analysis of Requirements Evolution in Open Source Projects: Recommendations for Issue Trackers" [98].

2016 Eloranta et al.: "Exploring ScrumBut—An empirical study of Scrum anti-patterns" [53] (extended from Eloranta et al. [54], with structural inspiration from Brown et al. [37] and Sutherland et al. [217]).

2016 Tamburri et al.: "The Architect's Role in Community Shepherding" [218].

2020 Telemaco et al.: "A Catalogue of Agile Smells for Agility Assessment" [222] (extended from Telemaco et al. [221]).

2022 Qamar et al.: "Taxonomy of bug tracking process smells: Perceptions of practitioners and an empirical analysis" [188] (extended from Qamar et al. [187]).

The second set of articles from which I extracted Best Practices is articles that mention some kind of quality attribute for ITSs. I list them here, in chronological order of publication date:

2006 Halverson et al.: "Designing Task Visualizations to Support the Coordination of Work in Software Development" [94].

2009 Aranda and Venolia: "The secret life of bugs: Going past the errors and omissions in software repositories" [10].

2010 Zimmerman et al.: "What Makes a Good Bug Report?" [250] (extended from Bettenburg et al. [27]).
2023 Prediger: "Visualising Data and Best Practices in Jira Issue Repositories" [184].
2023 Lüders: "Mining and Understanding Issue Links Towards a Better Issue Management" [137].

Methodological Considerations

Scope of ITE Best Practices. With the inclusion of certain articles above, the scope of what is considered an "ITE" Best Practice is called into question. ITSs are complex tools that support and manage many SE processes, and therefore there is a significant overlap of concepts that involve ITSs. What differentiates an ITE Best Practice from a Best Practice that only applies to Scrum (for example), is that there is some meaningful aspect of the Best Practice that involves an ITS. For example, the Scrum Best Practice "Product Owner is not Customer" [53] is not an ITE Best Practice because this is a role-based process recommendation that doesn't involve ITSs in any meaningful way. The Scrum Best Practice "Ordered Product Backlog", however, *is* an ITE Best Practice because multiple aspects of the process, recommendation, and benefits all involve an ITS in some way. Another way of looking at this overlap issue is to consider the fundamental purpose of an ITS: to support SE processes. All ITE Best Practices will support some SE processes. Therefore, all ITE Best Practices are also Best Practices for some SE process, such as Scrum. However, just because ITE Best Practices are also SE Best Practices, the opposite is not always the case (as described and illustrated above).

Extrapolating Beyond Explicit Statements The process of structuring these Best Practices involved extrapolating beyond the explicit statements made in the articles. Quotations have been used in the catalogue items where exact information has been copied. Other, non-quoted information is created using deductive thinking, applying the collective knowledge of this thesis to act as a guide for completing each catalogue item. Best Practices with less quoted information required more extrapolation, and require closer attention by future research.

Empirical Evidence in this Initial Catalogue. It should be emphasised that the listed ITE Best Practices are merely the structuring of statements from other researchers, this thesis provides no further evidence that these objects hold under the conditions stated. While the purpose of the Best Practice Ontology for ITEs is to increase the value of Best Practices through structure and guidance, it can also make the information appear more trustworthy or rigorous. This thesis does not provide any additional evidence that should make these Best Practices more trusted. However, the explicit structure should make it easier for researchers to identify which Best Practices require more evidence, and in what way (which dimensions, for example). Additionally, it should be emphasised that most of the research that derived the catalogue items does not provide direct empirical evidence of the benefits of the Best Practices, or the harmfulness of the violations. Most of the research presents analytical arguments based on Software Process Models (SPMs) being presented as SE theory. While there is a lot of value in

this approach, it does not provide direct empirical evidence that the Best Practices, embedded in the stated context, will provide the stated benefits. This is also true of research into ITE smells, which either extrapolate from SPMs, or gather opinions and insights from practitioners through interviews. Neither provide causal evidence of the impacts of these smells, violations, and Best Practices. Despite these considerations, this catalogue provides considerable value to SE researchers and practitioners. The structuring of these Best Practices exposes these considerations, and offers a path forward for structuring future evidence.

The Use of "None" and "Unknown" in the Catalogue. In the catalogue, the values for the dimensions will occasionally be either "None" or "Unknown". "None" represents that there is currently no known value for this dimension. This is quite common for both Inclusion and Exclusion Factors. In the initial formation of the Best Practices in this catalogue, however, "None" should be interpreted with care. It is not clear whether the information is simply not listed in the article, does not exist in the opinion of the original authors, or perhaps does not yet have any empirical evidence to warrant listing such information. I have done my best to extract meaningful insights out of the articles to fill in the Best Practices, but at times much of the information was missing. In cases where it was really not clear what should go in a particular dimension, I then used the value of "Unknown". In particular, "Unknown" was listed in dimensions where there should be a value, based on the Best Practice and the article describing it, but I was unable to find or deduce the information from the article. For example, there should always be at least one Stakeholder Benefit (otherwise the Best Practice has no reason to exist), but I could not deduce a meaningful stakeholder benefit for "BP12: Assignee Bug Resolution" (see Table A.13). I decided to leave this Best Practice in the catalogue, despite this missing information, for the exact purpose of highlighting cases like this. It is not clear what information should go here (it is "Unknown"), which is evidence that perhaps this Best Practice is not yet well-formed enough to be formulated and disseminated as a "Best Practice".

7.1.2. Validating the Catalogue: Interviews with Practitioners

In the same set of interviews reported in Chapter 3, I asked the interviewees about their perception of a set of ITE smells. To summarise the important methodological components, I interviewed 26 practitioners working in Germany, Canada, and Poland (full details listed in Table 3.1). They had a range of work experience from 1.5–25 years (median 7 years). They held various roles such as Developer, Manager, Product Owner, Solution Architect, and Consultant. All participants used an ITS in their current position and had a range of experience using different ITSs throughout their careers. The majority (14/26) primarily used Jira, with six more participants having experience with Jira. Others used GitHub [83], GitLab [84], Trac [229], Azure [14], Bugzilla [38], Trello [230], asana [13], Mantis [148], SpiraTest [214], RedMine [191], BaseCamp [18], and Miro [156]. The size of their ITSs ranged from just a few hundred up to over a million issues. The company sizes ranged from tens to thousands of employees, covering

different industries including automotive, medical, consulting, and energy.

In the interviews, I chose to ask the practitioners directly about smells (instead of violations or Best Practices) due to the approachability of the concept. While feedback on the entire concept of ITE Best Practices is desirable, I deemed the concept too complex and broad to be explained and critiqued in a single interview setting. Smells, on the other hand, are well-described, understood, and already utilised within industrial settings (whether for ITSs or tangential reasons such as code quality, as discussed in Section 2.3). The Best Practice Ontology for ITEs has five sections, one of which is "Violation", with the first dimension being "Smells". The Violation section represents the opposite behaviour or system state than is expected from the Best Practice. While the framing of a Best Practice is intentionally positive, and not negative (see Proposition 3 in Section 6.3), validating the existence and problematicness of the smells (violations) provides strong evidence towards the Best Practice overall.

The goal for interviewing the participants about the ITE Best Practice smells was to get their perception on occurrence and problematicness. I showed them a list of pre-collected ITE smells, and asked them "have you observed this smell" and "do you think it is problematic". The list of smells used is based on a subset of the ITE Best Practices in the catalogue. I explain below in more detail which smells were selected, and why.[1] Two data analysis methods were applied to the interview notes: Thematic Analysis, and a closed-labelling process.

We first conducted a Thematic Analysis of the notes to produce the qualitative findings. To gain an overview of the responses towards the ITE smells, we then conducted a closed-coding analysis of the RQ2 interview notes, in collaboration with another researcher. This was possible due to the trivial nature of the questions for each smell: "have you observed this smell" and "do you think it is problematic".[2] We did this by coding their responses into one of four mutually exclusive categories: yes, no, depends, or no answer. We first coded the responses separately, and then met to resolve the disagreements. Additionally, we applied the "depends" label when the participant explicitly stated that there are some situations in which the smell is problematic, and others where it is not (although the reason was not always stated). The question of "occurrence" does not warrant the "depends" code, since the participant either experienced the smell or not. We coded "no answer" when the participant had no comment, or their response was not conclusive enough to be coded into one of the other three categories.[3]

7.2. The Catalogue of Best Practices

The catalogue consists of 40 ITE Best Practices. In this chapter, I will describe the catalogue overall, the groups of Best Practices, and I will go into detail regarding a few key Best Practices. Given the space consumed by each Best Practice, the full catalogue is listed in the Appendix A,

[1] The list of smells is easier to understand, in context, once introduced to the entire ITE Best Practice catalogue.
[2] These are summarised; see our interview protocol for original questions.
[3] For example, one participant said, "it is interesting" in response to one of the smells, and nothing further.

instead of directly in this chapter. In both this chapter and in the appendix, there is a table of contents for the ITE Best Practices, to accommodate a succinct overview and quick lookup. The table of contents, Table 7.1 (and Table A.1 in the appendix), presents the Best Practices in three groups: Issue Properties, Issue Linking, and Issue Processes. Issue Properties Best Practices involve issue fields. Issue Linking Best Practices involve the linking of issues to other issues. Lastly, Issue Processes Best Practices involve ITS processes, including Scrum and generic Agile principles such as the backlog and sprints.

Table 7.1.: Best Practices Catalogue - Table of Contents.

ID	Name	Objective	Source	Page
		— Issue Properties —		
BP01	Good Bug Report	Achieve good Bug Report quality by having the necessary elements.	[27, 250]	179
BP02	Atomic Feature Requests	Each Feature Request has only one request.	[98]	180
BP03	Assign Bugs to Individuals	Achieve good Bug Report quality by having the correct fields set.	[187, 188]	181
BP04	Sufficient Description	Provide a sufficiently long description.	[184]	182
BP05	Succinct Description	Issue descriptions should be succinct, particularly for requirements and development issues.	[137]	183
BP06	Avoid Status Ping Pong	Streamline issue state changes through the identification and removal of state cycles.	[10, 94, 184]	184
BP07	Avoid Assignee Ping Pong	Streamline issue assignee changes through the identification and removal of assignee cycles.	[10, 94, 184]	185
BP08	Set Bug Report Assignee	Achieve good Bug Report quality by always setting an assignee.	[184, 187, 188]	186
BP09	Set Bug Report Priority	Achieve good Bug Report quality by always setting a priority.	[187, 188]	187
BP10	Set Bug Report Severity	Achieve good Bug Report quality by always setting a severity.	[187, 188]	188
BP11	Set Bug Report Environment	Achieve good Bug Report quality by always setting the environment variables.	[187, 188]	189
BP12	Assignee Bug Resolution	The bug assignee should be the one to resolve the bug.	[187, 188]	190
BP13	Avoid Zombie Bugs	Foster a meaningful backlog by keeping issues alive or resolving them.	[94]	191
BP14	Active Bug Reports	Cultivate a meaningful open Bug Report backlog through regular activity or archiving.	[10, 187, 188]	192
BP15	Bug Report Discussion	Achieve good Bug Report quality by encouraging discussion.	[187, 188, 218]	193
BP16	Respectful Communication	Maintain respectful discourse within issues and their comments.	[137]	194
BP17	Consistent Properties	Issue properties should always represent the most up-to-date information.	[137]	195

(continued on next page)

Table 7.1, continued

ID	Name	Objective	Source	Page
BP18	Good First Assignee	Achieve good Bug Report quality by assigning the best developer first.	[187, 188]	196
BP19	Stable Closed State	Increase the confidence that once a bug is closed, it will stay closed	[187, 188]	197
BP20	Timely Severe Issue Resolution	Address severe issues within a defined timeframe.	[94, 184]	198
BP21	Issue Creation Guidelines	Guide users how to create good issues.	[98]	199
BP22	On-Topic Discussions	Maintain topic-related discussions on Feature Requests.	[98]	200
		— Issue Linking —		
BP23	Bug-to-Commit Linking	Achieve good Bug Report quality by fostering traceability between bugs and resolving commits.	[187, 188]	201
BP24	Link Duplicates	Maintain good Bug Report quality by referencing related closed bugs.	[187, 188]	202
BP25	Minimal Link Types	Reduce redundant link types to simplify linking.	[137]	203
BP26	Record Links	Record all issue links using the linking feature, not in comments or in the description.	[137]	204
BP27	Realistic Dependencies	Catch and resolve unrealistic dependencies in link networks such as cycles.	[137]	205
BP28	Singular Relationships	Simplify ITS dependencies by limiting 1-to-1 relationships to a single link.	[137]	206
BP29	Connected Hierarchies	Connect hierachies with links to view the full picture.	[137]	207
BP30	High-Dependency Bugs First	Foster a low-dependency ITS.	[94]	208
BP31	Search Reminders	Decrease the number of duplicate Feature Requests by prompting users to search for existing ones first.	[98]	209
		— Issue Processes —		
BP32	Ordered Product Backlog	Maintain an ordered product backlog for meaningful product direction.	[53, 54]	210
BP33	Team-Produced Work Estimates	Create accountability between teams and their tasks by allowing them to make their own work estimates.	[53, 54]	211
BP34	Story Points Over Hours	Manage expectations and work towards correct predictions of work effort required per sprint.	[53, 54]	212
BP35	Estimate all Items	Estimate all work items before starting a sprint.	[221, 222]	213
BP36	Avoid Unplanned Work	Focus on planned work during a sprint, to achieve the agreed commitment.	[221, 222]	214
BP37	Recommended Sprint Length	Maintain a core element of Scrum by using 2–4 week sprints.	[53, 54]	215
BP38	Consistent Sprint Length	Maintain constant and simplified velocity by using the same sprint length for every sprint.	[53, 54, 221, 222]	216

(continued on next page)

Table 7.1, continued

ID	Name	Objective	Source	Page
BP39	Use Acceptance Criteria	Improve user stories by using acceptance criteria as a measure of completion.	[184]	217
BP40	Limit Acceptance Criteria	Improve user stories by limiting the number of acceptance criteria.	[184]	218

The full catalogue of 40 ITE Best Practices is listed in Appendix A, but here I will go into detail explaining the following Best Practices: Good Bug Report, Bug-to-Commit Linking, Consistent Properties, and Active Bug Reports (Avoid Zombie Bugs). All five of these Best Practices are from the Issue Properties group. This is an intentional decision, as Lüders [137] and Prediger [184] already discussed the Issue Linking and Issue Processes Best Practices in great detail. Neither of their works structure those Best Practices in the ontological form as my work, but they do focus on them as a key contribution of their work. Accordingly, I have structured those Best Practices into their respective groups of the ITE Best Practices catalogue, but I do not go into any further detail.

7.2.1. Formalising a Well-Researched Concept: Good Bug Report

I formed the "Good Bug Report" Best Practice from the seminal work by Zimmerman et al. [250] and Bettenburg et al. [27]. This is one of the earliest examples of strong empirical evidence being collected on a quality attribute of an ITS. Their primary objective was to understand what makes a good Bug Report from the perspective of developers. Their conference article by Bettenburg et al. [27] and their follow-up journal article by Zimmerman et al. [250] provide rich detail regarding their objectives and findings.

I chose this Best Practice as a starting place in this chapter because it is a well-known work, with solid empirical findings, that was relatively easy to extract into the ITE Best Practice format. This is an example of how the ontology can be used to structure existing strong research findings into a format for communication and comparison. Thus, I formed the "Good Bug Report" Best Practice in Table 7.2 (and in Table A.2 in the Appendix).

The interesting thing about this ITE Best Practice is what is not yet included in the table: a detailed analysis of all follow-up works on Bug Report quality, to transform the Best Practice in its current form into a much more rigorously detailed Best Practice. I created this Best Practice from the work of Zimmerman and Bettenburg alone, but there have been many follow-up articles regarding what makes a good Bug Report. Those additional works add evidence towards or against certain claims and recommendations presented in the Best Practice. Given the extent of the research in this area, there is enough evidence to make conclusive claims and prescriptions for practitioners. The Best Practice Ontology for ITEs can offer support in structuring and delivering this information to practitioners.

Table 7.2.: BP01: Good Bug Report.

Summary

Objective: Achieve good Bug Report quality by having the necessary elements.

Motivation: "Bug reports are vital for any software development. They allow users to inform developers of the problems encountered while using a software. [...] However, Bug Reports vary in their quality of content; they often provide inadequate or incorrect information."

Recommendation

Process: Bug reports should contain the elements most helpful for developers, which are (in order of helpfulness: steps to reproduce, stack traces, test cases, observed behaviour, screenshots, and expected behaviour.)

ITS: Provide a Bug Report template that must be used when submitting a Bug Report.

Context

Stakeholder Benefits: *Developers*: Get the information they need. *Reporters*: Get their reports resolved faster.

Stakeholder Costs: *Reporters*: Have to put more effort into submitting the required information.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: OSS where the reporters are external to the development organisation.

Exclusion Factors: None

Violation

Smells: Bug reports that consist of entirely unstructured text with no sections or headers.

Consequences: "Developers are slowed down by poorly written Bug Reports".

Causes: Reporters don't know what information to provide. Reports don't want to put in the time to provide all required elements. Reporters don't know how to get all the required elements.

Algorithmic Detection: Automate the detection of missing elements, and recommend that they be added before allowing the user to submit the Bug Report.

Extracted and extended from Zimmerman et al. [250] and Bettenburg et al. [27]

Table 7.3.: BP23: Bug-to-Commit Linking.

Summary

Objective: Achieve good Bug Report quality by fostering traceability between bugs and resolving commits.

Motivation: "To obtain useful information about a software project's evolution and history." Bugs are resolved by committing code that resolves the reported problem. If a bug is revisted in the future, commit linking allows them to quickly navigate to the correct piece of code. Additionally, these links can be used in reverse to understand why a commit was conducted, either for reasons of insufficient commit messages, or building automated support for forwards and backwards traceability.

Recommendation

Process: "[...] all bug-fixing commits should be linked to their respective Bug Reports." Before a bug is marked as "resolved" and therefore closed, the associated code commit should be conducted, and the link to that commit should be pasted into the associated Bug Report.

ITS: Have a dedicated "Commit" field that is mandatory before the Bug Report can be marked as "Resolved".

Context

Stakeholder Benefits: *Developers*: When revisiting bugs to learn or trace what happened, they will know what code was implemented. *Testers*: They know what code was added based on a Bug Report.

Stakeholder Costs: *Assignee*: Has to add the link to the commit in the Bug Report.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: The organisation chooses not to link their ITS with GIT.

Violation

Smells: "If a bug is closed without any link to the bug-fixing commit, then in the future, it will be difficult to discover what happened with that bug. We evaluate this lack of traceability as a process smell and we call this No Link to Bug-Fixing."

Consequences: "The potential impact of the [violation of this Best Practice] is losing track of the bugs and eventually the traceability of the bug decreases. It also affects the related software development tasks such as prediction of bug locations, recommendation of bug fixes, and software cost."

Causes: "Developer forgets to mention, Committing link is not straightforward in [ITS], Weak understanding of [ITS]"

Algorithmic Detection: "First, we check whether the bug is fixed. If so, we check the comments and designated fields to find a link to the version control system."

Extracted and extended from Qamar et al. [187, 188]

7.2.2. Offering a Clear and Simple Solution: Bug-to-Commit Linking

I formed the "Bug-to-Commit Linking" Best Practice from the work of Qamar et al. [187, 188]. This is a popular concept in practice, that has received moderate attention in research [15, 28, 187, 188]. The primary objective with this Best Practice is to foster traceability between bugs and their resolving commits. Table 7.3 showcases this Best Practice.

This Best Practice is a good example of a clear and simple "ITS Recommendation": "Have a dedicated 'Commit' field that is mandatory before the Bug Report can be marked as 'Resolved'." The previous Best Practice ("Good Bug Report"), can be automatically checked for violations using NLP and conformance templates; however, the certainty with which it can be detected, the preferences and ambiguities related to natural language descriptions, and the capabilities of the stakeholders managing the ITS all play a major role in how well the violations can be detected. For Bug-to-Commit Linking, violation detection is 100% accurate and easy to implement. If an organisation decides they want to adopt this Best Practice, there is no barrier for them to implement its automation.

7.2.3. Establishing a Best Practice Using the Ontology: Consistent Properties

I formed the "Consistent Properties" Best Practice from the work of Lüders [137]. The objective of this Best Practice is to maintain up-to-date, consistent properties within issues, attempting to combat the habit of ITS stakeholders who add properties information to the description or comments section, without updating them in their correct place. For example, when someone adds a comment that this issue is a duplicate of another, but they do not add the "duplicate" link connecting the two issues. This leads to issue properties that are inconsistent with the up-to-date information available on this issue. Table 7.4 showcases this Best Practice.

While Lüders mentions this Best Practice in a list of smells, she goes into no further detail. This ITE Best Practice is an example of a initial idea from someone, that is first established within the structure of the Best Practice Ontology for ITEs. I used the ITE Best Practice Ontology, and my tacit knowledge of ITEs, to deductively fill in the entire Best Practice. The result is just enough information to communicate the Best Practice for its intended purpose. Now that the ITE Best Practice has been formed, it is open to scrutiny, and hopefully eventually empirical evidence to strengthen or refute the claims made.

7.2.4. Surfacing and Amending Duplicate Work

I formed the "Active Bug Reports" Best Practice from the work of Qamar et al. [187, 188] and Aranda and Venolia [10], and I formed the "Avoid Zombie Bugs" Best Practices from the work of Halverson et al. [94]. The objective for both of these Best Practices is to cultivate meaningful open Bug Reports through regular activity or archiving. They share the same objective, and indeed they are essentially the same Best Practice, except one is framed from the positive

Table 7.4.: BP17: Consistent Properties.

Summary

Objective: Issue properties should always represent the most up-to-date information.

Motivation: Issues are both a form of structured information, and unstructured dialogue. This means that it is possible, and indeed often the case, that information in the description or comments contradicts information listed in the properties. For example, someone comments that this issue is a duplicate of another issue, but does not add this as a "duplicated by" link in the properties. Over time, this leads to inconsistencies between the properties and the dialogue.

Recommendation

Process: Update properties instead of updating the description or adding a comment.

ITS: Automate the detection of properties in the description and comments, and warn assignees and managers about the existence of such properties. This can be done with some certainty, but natural language of course makes this a task that comes with uncertainty.

Context

Stakeholder Benefits: *Everyone*: Can trust the properties as being up-to-date.

Stakeholder Costs: *Everyone*: Must actually update the properties. *Privileged Users*: Must update properties after non-privileged users add comments with new information.

Artefact Scope: Issue

Issue Types: All

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: Properties are mentioned in the comments or description.

Consequences: The entire issue must be read in order to understand the current state, and ultimately to trust the properties as they are set. While this is possible for any single issue, this is not manageable for the entire ITS.

Causes: People may not have the permission to update the issue itself, so they make a comment instead. People may not know that there is a property for this information. People might be lazy, and prefer to make a quick comment.

Algorithmic Detection: Search the text for potential property updates. This includes searching for the names of the properties, as well as all states the properties can be in. For example, search for the word "status", but also search for "Open" and "Resolved". With some additional NLP intelligence, the false positives can be reduced to a manageable low number.

Extracted and extended from Lüders [137]

Table 7.5.: BP14: Active Bug Reports.

Summary

Objective: Cultivate a meaningful open Bug Report backlog through regular activity or archiving.

Motivation: "Bugs that are left open for a long time or bugs that have incomplete resolution. In a [bug tracking system], once the bug is opened, it should not be left unattended or open for a long time. The knowledge of the bug may be forgotten over time. Even if it is not closed, some progress should be made to resolve the bug."

Recommendation

Process: Revisit Bug Reports that have not been updated in more than X months, and either perform a meaningful action towards resolution, or mark the Bug Report as "will not fix". The recommended maximum is 3 months, but a more reasonable limit (to maintain working knowledge) is 1 month.

ITS: Create a dashboard or notification system that contacts the relevant stakeholders when a Bug Report has not been updated in X months.

Context

Stakeholder Benefits: *ITS Users*: They know the open Bug Reports are meaningfully curated and represent live issues.

Stakeholder Costs: *Assignees*: Have to continuously assess whether it is worth keeping a Bug Report open or not.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: If the organisation maintains open bugs for future reference, i.e., a history of what was reported but never addressed.

Violation

Smells: Assignees ignoring issues and issues that are delayed for too long.

Consequences: "The potential impact of [the violation of this Best Practice is a] delay in the bug resolution process." Another potential impact is a growing backlog of open Bug Reports that may never be resolved.

Causes: "Incorrect severity indication, Inadequate bug description, Incorrect prioritization, Overlooked bug."

Algorithmic Detection: "We compare the dates of sequential activities in the bug history. While the bug is not resolved, if there is a gap longer than three months between any two activities, we consider it a [violation]."

Extracted and extended from Qamar et al. [187, 188] and Aranda and Venolia [10]

Table 7.6.: BP13: Avoid Zombie Bugs.

Summary

Objective: Foster a meaningful backlog by keeping issues alive or resolving them.

Motivation: "Zombie bugs are defects that have lain dormant for a (relatively) long period of time. Low priority bugs that turn into zombies may not be a problem. However, being aware of zombie bugs was reported as an important part of project housekeeping."

Recommendation

Process: Regularly check how long bugs have been open, tending to those older than a certain time. Decide as an organisation how long bugs should remain open before they are set to "won't fix", or something similar.

ITS: Automate the resolution of bugs older than a certain amount of time. The resolution will be some kind of "won't fix", not the standard "fixed".

Context

Stakeholder Benefits: *Managers and Developers*: Have a meaningful backlog where all open bugs should be worked on.

Stakeholder Costs: *Managers and Developers*: Need to address these older bugs by either working on them, or changing the status to resolved. This cost is eliminated with the automated approach to marking bugs as resolved.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Defects [(Bugs)] that have lain dormant for a (relatively) long period of time."

Consequences: Backlogs grow indefinitely if not otherwise addressed or resolved, which leads to a lack of meaning to the status "open".

Causes: Developers forget about bugs.

Algorithmic Detection: 1) Search for all bugs older than X days (X defined by the organisation). 2) Either mark the bug as "won't fix" or "stale" as a resolution, or notify the assignee that they need to either act on the bug or change the status themselves.

Extracted and extended from Halverson et al. [94]

("Active"), and one is framed from the negative ("Avoid").

I chose these two Best Practices to discuss here precisely because they are essentially the same Best Practice. When I first encountered these two Best Practices, I considered combining them; however, the task of synthesising ideas such as Best Practices is a complex and tedious one. The purpose of this catalogue is not to claim that these Best Practices are the reduced form of all possible Best Practices, and so I did not perform this task. The work of Qamar et al. [187, 188] from 2022 references the work of Halverson et al. [94] from 2006, but only in the related work section, and not in the formative section outlining the Best Practice. There is no clear scientific connection between the two specific Best Practices, and so they live in isolation from each other. I did not notice their connection until I formed them into ITE Best Practices, despite having a strong familiarity with both works. With an ontological structure such as the ITE Best Practice ontology, hopefully more such examples can be found and resolved.

7.3. Practitioner Feedback on the ITE Best Practice Smells

7.3.1. Selected ITE Smells

I selected 31 smells in collaboration with two other researchers who were also researching similar topics. My focus was on Issue Property smells, while the focus of the other researchers was on Issue Link smells and Issue Process smells. Table 7.7 lists the 31 smells, as well as the associated ITE Best Practices. The mapping from smell to ITE Best Practice is not one-to-one. A smell can have multiple Best Practices if the Best Practices are more granular than the smell, and vice versa. A smell can also have no ITE Best Practice if the smell was created or discovered independently of the Best Practice catalogue. This is the case for many of the Issue Link and Issue Process smells, since they were created independently of my research. For the Issue Property category, the only smell without a Best Practice is "too many issue types", which was a smell we added just for the interviews, to compare with the findings from the "too many link types" smell. As most smells come from the ITE Best Practices, the origin of the smells can be traced to the articles described in Section 7.1.1. In particular, we asked about the occurrence of these smells, as well as their opinions on how problematic each smell is.

7.3.2. Results

I describe here the results of showing each participant our list of smells and getting their feedback on 1) have they experienced these smells, and 2) how problematic are they. I visualise the closed-coded smell responses in Figure 7.1. Each cell in the figure represents a response from one of the participants. The cells are across each of the 31 smells and 26 participants. The initial coding of responses showed 74% agreement ($\kappa = 0.61$), with a final agreement of 100%. I added four participant context factors from Table 3.1 to the left side of Figure 7.1 to

Table 7.7.: List of smells discussed in the interviews.

	ID	Smell	ITE Best Practice (ID [Table] Name)
Issue Property Smells	S1.1	No or short (non-informative) description	BP04 [A.5] Sufficient Description
	S1.2	Description too Long	BP05 [A.6] Succinct Description
	S1.3	Issues are missing properties (assignee, priority, severity, ...)	BP08 [A.9] Set Bug Report Assignee BP09 [A.10] Set Bug Report Priority BP10 [A.11] Set Bug Report Severity BP11 [A.12] Set Bug Report Environment
	S1.4	Issues are assigned to a team	BP03 [A.4] Assign Bugs to Individuals
	S1.5	Often switching properties (status or assignee, ...)	BP07 [A.8] Avoid Assignee Ping Pong BP06 [A.7] Avoid Status Ping Pong
	S1.6	Too many issue types	
	S1.7	No link to commit	BP23 [A.24] Bug-to-Commit Linking
	S1.8	Non-assignee resolved issue	BP12 [A.13] Assignee Bug Resolution
	S1.9	Ignored issue/delayed for too long	BP14 [A.15] Active Bug Reports BP13 [A.14] Avoid Zombie Bugs
	S1.10	No comments or too many comments	BP15 [A.16] Bug Report Discussion
	S1.11	Toxic discussions	BP16 [A.17] Respectful Communication
	S1.12	Properties discussed in comments but not updated in issue	BP17 [A.18] Consistent Properties
Issue Link Smells	S2.1	Issue without any links	
	S2.2	Too many link types / Link types with overlapping meaning	BP25 [A.26] Minimal Link Types
	S2.3	Multiple link with differing types to the same issue	BP28 [A.29] Singular Relationships
	S2.4	Known link mentioned in comments but not documented	BP26 [A.27] Record Links
	S2.5	Mismatch between link types and properties	
	S2.6	Mismatch between linked issues regarding their status, due date, priorities, or estimates	
	S2.7	Epic without sub-issues or sub-issues without main-issue	BP29 [A.30] Connected Hierarchies
	S2.8	Circular dependencies between issues	BP27 [A.28] Realistic Dependencies
Issue Process Smells	S3.1	Unplanned work added during sprint	BP36 [A.37] Avoid Unplanned Work
	S3.2	Too many complex issues are assigned to the same sprint	
	S3.3	Issues are missing an estimate	BP35 [A.36] Estimate all Items
	S3.4	Estimate-scales differ between issues/ unclear estimate-scales	
	S3.5	Sprint does not end at scheduled time	BP38 [A.39] Consistent Sprint Length
	S3.6	Sprints have different duration	BP38 [A.39] Consistent Sprint Length
	S3.7	Sprint length differs from recommended length	BP37 [A.38] Recommended Sprint Length
	S3.8	Sprint has to be (repeatedly) delayed	
	S3.9	No acceptance criteria or too many	BP39 [A.40] Use Acceptance Criteria BP40 [A.41] Limit Acceptance Criteria
	S3.10	Acceptance criteria are not checked during testing	
	S3.11	Acceptance criteria are changed during sprint	

make it easier to cross-reference them. The bottom and right sides of the figure include the counts of responses for each smell and participant.

All the smells were noted as occurring by at least some participants. Most of the smells were also noted as problematic by the participants, but 98 of the 290 responses were either "no" or "it depends". Given the lack of contradictory evidence in research, these statements are the most interesting. For this reason, I will focus the analysis on these responses.

Issue Property Smells

Participants largely agreed that Issue Property smells occur, but 28 of their responses said they are unproblematic, and 31 responses noted a related context factor. Here I discuss those smells and the stated context factors in decreasing order of disagreement.

Issues are assigned to a team (S1.4) received the most disagreement of the Property (8/9 participants that answered). Most of the reasons for disagreement mentioned that issues were intentionally assigned to teams as part of their workflow. P01 and P26 mentioned that issues are assigned to a team first, then to an individual. P01 and P24 said that issues are assigned to teams to "engage" the teams, with the main assignment left to them to "decide autonomously". P12 said that team assignments can be used for downstream activities such as "in review", whereby the team responsible for reviewing the issue will be assigned. Finally, P14 mentioned that this kind of workflow dynamic "depends on the team".

Non-assignee resolved issue (S1.8) received many disagreements (7/10 participants). Many participants mentioned that non-assignees resolving the issue is an explicit part of their process. For example, P01 mentioned that he is closing tickets as part of his role, regardless of the assignee. P06 said that issues are often resolved as part of a group meeting. P26 mentioned that it only happens when the original assignee cannot work on it, so the assignee is changed, but the original assignee could still change the stated to "resolved".

Issues are missing properties (assignee, environment, priority, severity, ...) (S1.3) also received many disagreements (9/15). Three participants mentioned that fields may be blank at the beginning, but should later be filled. P15 and P16 both said that fields such as the assignee and weight of the issue are assigned later, such as on the first day of the sprint. P01 mentioned something similar, remarking that "no assignee while the ticket is in progress would be considered bad". Participants also mentioned that it depends on the field, since fields such as Severity are important and cause for concern if missing (P26). P16 said it depends on the field, and often the field is implied (known tacitly, given the context). P22 mentioned that missing properties are "not so important in small team settings, more in large settings". Interestingly, P06 and P24 mentioned that some fields are enforced by the ITS. This means that they are so problematic if left empty, that the companies have put a technological barrier in place to prevent them from being blank.

No comments or too many comments (S1.10) had 9/19 disagreements. The primary dis-

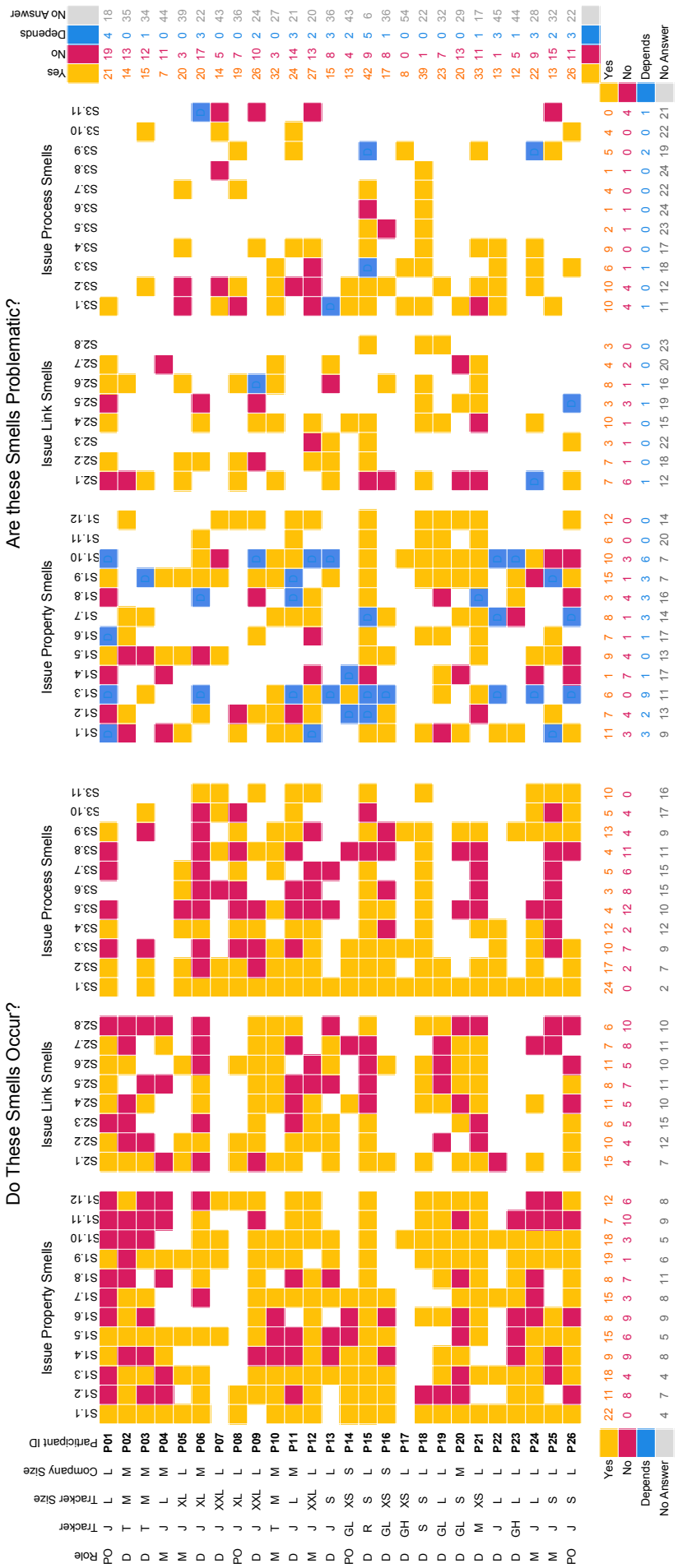


Figure 7.1.: Participant responses on whether the smells occur (left), and whether they think they are problematic or not (right). Green: Agree, Red: Disagree, Blue: It Depends, Blank: No Response.

agreement about this smell is the number of comments considered bad. P01 said he doesn't consider too many comments bad, but he also thinks that "over 100" would be a problem. P07 and P26 both commented that "no comments is not a smell", with P26 elaborating "because the daily [standup] should be used for that, [which then leads to] a description change". Others mentioned the number of problematic comments depends on the issue (P12), with long comments important to their role (P12), and "unimportant stuff is the most problematic" (P13). P09 doesn't think it is a smell, but he also admitted that he "doesn't want to read them all".

Description too long (S1.2) was another smell with split opinions (6/13). Participants mentioned that long descriptions offer more information, which they value. For example, P01 said long descriptions offer a "better impression", P08 values longer and more complex descriptions, and P11 said, "the more information the better". Other participants said that it depends on the issue type. P14 said it is "context-dependent" since long descriptions are not good for small Bug Reports, but epics often require longer descriptions. P15 mentioned that long descriptions can be a problem, but noted that "complex issues" may require long descriptions.

No or short (non-informative) description (S1.1) had 6/17 in disagreement. The main reason participants said a short (or empty) description was not problematic was when the information could be found elsewhere. P01 said that no description is usually bad, "but there are [issue] types where a title is sufficient". P12 said that there is sometimes a comment that mentions external information, such as a Pull Request. P04 mentioned that it is often sufficient if the issue has a Parent issue. This hierarchical structure of issues can lead to well-defined Epics, with children User Stories that just contain a title and no description. Participants P25 and P04 mentioned that "too short" is difficult to define. P25 said, "we like shorter descriptions, but single words can also be hard to interpret or remember". One participant (P01) said that "it depends on the context" without further qualifier. Finally, P12 also said that short descriptions "are often not an issue, if the issue is created by me and assigned to me".

No link to commit (S1.7) had 4/12 in disagreement. P15 said that it depends on the issue type, since issues such as support requests don't have to be linked to anything. P26 mentioned that it only matters if this issue and its link need to be tracked. Participants mentioned that they had set up automation in their ITSs that automatically add a commit link when the issue is mentioned in the Git commit. This is also the case for trackers such as GitHub that are already linked with the Git system. These automations highlight the importance of linking.

Often switching properties (status, assignee, ...) (S1.5) had 4/13 in disagreement. P04 said that the assignee switching back-and-forth might be used by people "to keep their statistics clean" (fewer issues assigned to them). It was also, however, used as an intended process similar to peer-review. For example, P02 said it is "part of their process to pass the issue back and forth between assignee and reviewer [as] their quality assurance".

Ignored issue/delayed for too long (S1.9) only had 4/19 in disagreement. P03 mentioned that it is only a problem if the release is affected by it. P11 said, "old backlog issues are often not important". From an information perspective, P24 said the "backlog is cared for" because these

ignored issues "contain knowledge" and therefore are important. P24 added that some issues "need some more time to ripen". P25 mentioned that how problematic these ignored issues are depends on the discipline of the team and the habits of interacting with the ITS.

Too many issue types (S1.6) had 2/9 in disagreement, but no explicit contextual factors were mentioned. The smells *Toxic discussions (S1.11)* and *Properties discussed in comments by not updated in issue (S1.12)* were stated to be problematic by all participants who gave a response (6 and 12, respectively).

Issue Link Smells

Participants largely agreed that Link Smells occur, but 15 of their responses disagreed that these smells are problematic, and 3 responses noted a dependent context factor.

Mismatch between link types and properties (S2.5) had the most disagreement with 4/7 participants, but very few mentioned context factors. P01 mentioned that if it happens, it tends to be with duplicate reports (wrong issue type of "duplicate" used instead of something else).

Issue without any links (S2.1) had many disagreements with 7/14 participants. P24 mentioned that if the issue is too small, then there is no need for links. P02 said that certain types of issues, like support requests, often do not have or need links.

Mismatch between linked issues regarding their status, due date, priorities, or estimates (S2.6) had 2/10 participants in disagreement. P09 said that can happen, but only the status fields being out of sync matters, not the resolution. Both P09 and P13 mentioned that this problem is largely solved by external communication regarding the issue.

Issue Process Smells

Participants largely agreed that Issue Process Smells occur, but 16 of their responses disagreed that these smells are problematic, and 5 responses noted a dependent context factor.

Acceptance criteria are changed during sprint (S3.11) had full disagreement with all 5 participants that responded. The consensus among the participants was that it is more important to update the acceptance criteria, than to leave it outdated when you know it should be changed. As P25 stated, "it is important to update the acceptance criteria if the change is needed".

Unplanned work added during sprint (S3.1) had 5/15 participants in disagreement. The five participants who disagreed all mentioned that unplanned work was a normal part of sprint operations. P08 said that it "happens every sprint" and is "normal operations". P12 emphasised that it is their "default" and represents standard procedure. P21 added that this is an important topic, and it is not a problem "as long as you get your work done".

No acceptance criteria or too many (S3.9) had 2/7 participants in disagreement. P25 said that it "depends on who is asking for the feature", and noted that if any problem exists, it is more about no acceptance criteria than too many. P24 said that it is not a problem since "no acceptance criteria happens, and they are created after the fact".

Too many complex issues are assigned to the same sprint (S3.2) had 4/14 participants in disagreement. P05 and P07 both said the smell is unproblematic because you can delay issues to the next sprint without repercussions. P11 said complex issue count is not enough, since it is not a problem as long as the complex issues are distributed between people correctly.

Issues are missing an estimate (S3.3) had 2/8 participants in disagreement. P12 said that they don't do estimates because of their very complex code base, which always leads to wrong estimates anyway. P15 mentioned that it depends on the sector you are working in, since not all companies have to agree on estimates ahead of time.

The following smells had disagreements, but no mentioned context factors: *Sprint has to be (repeatedly) delayed (S3.8)*, *Sprints have different duration (S3.6)* had 1/2 participants in disagreement, and *Sprint does not end at scheduled time (S3.5)*. The last three smells had no disagreements, with all participants who answered agreeing that the smells were problematic. Those smells are *Acceptance criteria are not checked during testing (S3.10)* (4 participants), *Sprint length differs from recommended length (S3.7)* (4), and *Estimate-scales differ between issues/ unclear estimate-scales (S3.4)* (9).

7.4. Related Work

ITE Smells. This new area is the closest to our work, with preliminary studies that presented the starting point for our RQ2. Aranda and Venolia [10] conducted a survey and a mining study to extract coordination patterns such as "forgotten" and "close-reopen" bugs. Eloranta et al. [53] conducted semi-structured interviews to identify 14 agile "antipatterns". Recently, Qamar et al. [187] proposed a taxonomy of 12 bug tracking process smells. They surveyed 30 developers about these smells and mined their occurrences in 8 open-source projects [188]. Survey respondents also commented on actions they took to avoid the smells. They observed a considerable amount of the smells in all projects, and the majority of surveyed practitioners agreed with the smells. Our work covers 19 additional issue smells, including issue linking and process smells. We ran hour-long interviews to uncover when and in which context the smells actually may lead to problems and why. Finally, our goal is to understand the overall challenges in ITSs, including the smells and their management. Our results discuss possible confounding factors that may explain such correlations. Finally, Tuna et al. [232] studied the 12 smells by Qamar et al. at JetBrains, surveying 24 developers at this company. Similar to our results, they also found the perception of smell severity to vary across smell types, and that smell detection tools are considered useful for only six of the smells. We studied a larger catalogue of smells by interviewing 26 practitioners in 19 different companies.

Halverson et al. [94] observed collaboration antipatterns in ITSs. Tamburri et al. [219] defined and evaluated community smells [40] that might lead to unforeseen project cost due to a 'suboptimal' community. They used the term "community smell" as organisational and social circumstances which cause mistrust, delays, and uninformed or miscommunicated architectural

decision-making. Palomba et al. observed that community smells contribute to the intensity of code smells [175], which motivates our research on ITE smells.

There are some studies, mostly surveys, evaluating the perception of SE research among practitioners [39, 135, 251]. Zou et al. [251] conducted follow-up interviews with 25 survey respondents provided further insight into their perspectives. Zou et al. [251] surveyed 327 practitioners to understand better how practitioners view these techniques. The survey asked participants to rate the importance of various categories of automated Bug Report management techniques and provide their rationale.

7.5. Discussion

The findings highlight and confirm an important characteristic of ITE smells: their relevancy to a given set of stakeholders is *highly context-dependent*. Research on ITE smells up until this point has stated binary opinions on their applicability, usually related to a single easy-to-interpret contextual variable, such as "uses agile". For example, if a company "uses agile", these smells apply to their company. A similarly extreme—and incorrect—position to take would be to say that there are no shared context factors across interpretations of the smells, and every smell depends on the individual who interprets it. ITE Best Practices *apply across consistent context factors*, but much more research needs to be conducted to formally understand and label these context factors on a per-best-practice basis. For example, given the context factors "company uses acceptance criteria" and "acceptance criteria apply to issue types x, y, z", it is expected that the smell "no acceptance criteria or too many" applies. While individual developers may disagree on a per-issue basis, having that smell auto-detected within their ITS is likely going to raise the quality of the ITE process over time. Currently, our understanding and definitions of ITE smells are missing empirically grounded context factors.

Good and bad ITE practices strongly vary among the studied practitioners and their organisations. The results highlight that the perceived relevance and severity of 31 smells from the literature are context-dependent. What is considered risky for some was a deliberate, intended procedure for others. For instance, practitioners generally agreed that issues should not be ignored for too long, descriptions should be informative, and knowledge should not remain hidden in comments. However, smells like "issue ping-pong" [94], lack of comments on an issue, and issues being assigned to a team [187] were an intentional part of their workflows. This has several implications. First, teams and even single users should be able to configure specific smell detection and management approaches: not only what should be detected as smells but also at what threshold and for whom. Such configuration can be challenging for administrators, particularly at the setup time of the ITS. My work provides guidance for (a) what may matter in which context and (b) how to involve stakeholders in internal smell configuration study.

One interesting research direction is to "learn" the smells and their severity by analysing the interactions of stakeholders within an ITE. Such learned smells can assist ITS users externalise

and assess their practices and potential smells. Moreover, it remains unclear whether the perceptions of practitioners always reflects the objective practice. Therefore, analytical approaches that visualise the overall state of ITEs and the evolution of certain issues can be informative no matter whether a particular observation is an actual smell or not.

Interviewees discussed many broad problems they are experiencing while using ITEs, but very few specific problems with granular solutions. This highlights the need for more granular approaches to improving quality in ITEs—such as well-defined context-dependent smells. When asked about problems with their ITSs, large, unbounded, and vague problems surfaced, such as information overload, and workflow bloat. These are real and serious problems that affect the daily lives of SE practitioners, but they cannot be addressed directly. ITE smells, on the other hand, are designed to be specific units of potentially problematic situations, that can be automatically identified and explained to the user. For example, Zombie Issues are a real problem that is easy to explain to users and can also be formulated as a concrete smell, such that a script can automatically notify the correct involved stakeholders when it is detected at sufficient levels; however, if left unattended (not identified automatically or manually), this leads to a general sense of anxiety and uncertainty with the ITS, since there are hundreds (if not thousands) of ageing issues that clog up the system. This then further leads to additional problems such as searching difficulties (due to too many open issues), assignee ping-pong (to reduce personal accountability), and incorrect information in issues (since ageing tickets leads to loss of information). Well-defined smells, with specific context factors, can be a remedy for these large uncertainties ITE users face.

Finally, the results suggest that issue smell tools seem particularly useful for managers and product owners. This can, however, be due to certain misconceptions or biases, such as developers being "afraid" of control, tool scatter, and overhead. What makes smells particularly challenging is that their negative impact is often not immediate. Future lab or observational studies with developers can control potential biases and clarify how smells should be presented and explained to different stakeholders in the ITS. For instance, while issue structure visualisation could be helpful for navigating the knowledge graph around an issue, it may seem unnecessary or too complex for people who prefer list visualisation [133].

7.6. Summary

In this chapter, I formed 40 Best Practices for ITEs using an inductive approach. The resulting catalogue can be used by practitioners to apply these recommendations in practice. From a research perspective, researchers can now challenge and improve these Best Practices utilising a unified structure. One finding from the catalogue-forming process is that some ontological attributes are discussed more than others in existing literature. For example, the Violation section is heavily discussed in existing literature, while Context is quite rare. Therefore, future work needs to fill this gap with case studies focused on investigating and enhancing existing

Best Practices in industrial settings.

One core part of the Best Practices is their potential for automation. The Violation section explicitly lists "Algorithmic Detection" as one of the dimensions. Many of the formed Best Practices list either pseudocode or plain-text descriptions of how to algorithmically detect violations. One way to further enhance the usability and application of these Best Practices in industrial settings, is to make these algorithms a reality. In the next chapter, I implement these algorithms and apply them to my existing dataset of Jira Repos. The results showcase the prevalence of these violations in a real-world dataset of ITSs.

Chapter 8.

Automation of Best Practices for Issue Tracking Ecosystems

> Automation is driving the decline of banal and repetitive tasks.
>
> Amber Rudd

In this chapter, I demonstrate the application of algorithms to detect violations of Best Practices described in Chapter 7. Chapter 6 introduced the ontology, thereby providing the structure necessary to describe ITE Best Practices, and Chapter 7 then presented a catalogue of Best Practices formed into the ontology. These chapters, however, don't go into detail regarding the algorithmic detection of associated violations. While the ontology itself provides much value in structuring existing research and focusing future efforts, it does not require that all Best Practices have algorithmically detectable violations. Simply by using the ontology as a guide, it is possible for an organisation to examine their ITEs, reveal latent Best Practices, document them, and even start improving them by deploying manual processes. However, it is much more desirable to have algorithmically enabled processes to automatically (or semi-automatically) detect violations of Best Practices. Using these algorithms, practitioners can preemptively prevent issues from spreading within their ITSs and beyond.

To provide such algorithmic detection techniques, I conducted a mix of related work extraction and novel algorithm creation. Some related work on ITS "smells" and "Best Practices" has provided algorithmic detection methods, in which case I cite and describe their work. In other cases, I have developed novel detection algorithms. I do not offer an exhaustive systematic list of algorithmic approaches, but rather an initial set of algorithms to show the feasibility and diversity of techniques available to support ITE Best Practices. As a result of this algorithmic exploration, I describe 18 algorithms that I designed or extended from the Best Practices. Twelve of those algorithms are specific to Bug Reports, and 6 apply to all issue types. This chapter contributes a concrete set of algorithms to detect violations of Best Practices, such that research can improve on them, and industry can apply them.

This chapter completes Part III Solution Construction, in combination with the previous Chapters 6 and 7. The next chapter starts Part IV Outlook Recommendation, which reflect

on the previous chapters and further apply them. In particular, the next chapter (Chapter 9) builds on the concepts of individual algorithms for violation detection, and looks at the greater picture of tooling for ITE Best Practices.

Publications. This chapter contains portions of my collaborative publication, some copied verbatim [165].

8.1. Research Methodology

My primary objective with this chapter is to demonstrate the automation of ITE Best Practices. For this, I have a singular research question:

RQ1 To what extent can ITE Best Practice violations be automatically detected?

To investigate this research question, I explored the literature discussed in Chapters 6 and 7, searching for algorithms. Most of them describe detection techniques in at least pseudocode, while others say nothing. As a result of this search, I implemented 18 algorithms. To showcase the algorithmic detection of violations to ITE Best Practices, I applied these algorithms to my Jira dataset presented in Chapter 4. This dataset has 16 Jiras, but I only use 13 of them in this chapter, for the same reason as described in Chapter 5: two of the original repos (MariaDB and Mindville) contain no comments, and Mojang only contains bugs. To put that in a broader perspective for this chapter, presenting results from those repositories alongside the others would present a skewed perspective, since they are simply missing data that the others are not. This would not be clear from the figures presented in this chapter, and would therefore give a false representation of the findings across the repositories.

For each ITE Best Practice, I present the results of detecting violations in a consistent figure format. This format can be seen in Figure 8.1, which shows a summary of all ITE Best Practice violations related to Bug Reports within the dataset. Given the diversity of ways in which Jira is used, presenting a single value result (such as a global mean or median) would not provide meaningful insights into these ecosystems. Accordingly, the figures split the results across Jiras and Activities (from Section 4.3.1), presented as box plots across the distribution of projects at each intersection. Each box plot represents the percentage of issues that violate the Best Practice, across all projects for a given Jira.

To describe Figure 8.1 in detail, I will use the first row as an example: “BP08: Set Bug Report Assignee”. This row is plotting the percentage of violations to that Best Practice, which means that the “Assignee” field is not set on Bug Reports that are resolved and closed. Each column of data represents a single Jira Repo, and the box plot visualises the percentage of violations to the total number of issues per project. For example, the first column is visualising that the Jira Repo “Apache” has a median of 10% violations to this Best Practice, across all projects in that Jira Repo. In contrast, IntelDAOS has 0% violations of this Best Practice. Given the overall low violation rate, the y-scale is in symlog scale to highlight values in the 0–10% range. Other Best Practices, such as “BP11: Set Bug Report Environment”, have their y-scale in linear scale,

to better visualise the high violation rate. The "Overall" plots presented on the right side of the figure show the results across all Jiras (still as a per-project distribution).

For the summary figures (Fig. 8.1 and Fig. 8.13): each row is a Best Practice being applied to all relevant issue types (i.e. Bug Reports for Fig. 8.1 and all issue types for Fig. 8.13). For the individual Best Practice figures (e.g., Fig. 8.2 or Fig. 8.14), each row is an issue type, or issue type theme. This breakdown into issue type themes (where possible) allows an even more in-depth analysis of the violation rate for a given Best Practice.

For the 18 algorithms implemented in this chapter, there are two major categories: ITE Best Practices related to Bug Reports and those that apply to all issue types. This is not by design, but rather as a natural consequence of the SE ITS research focus on Bug Reports over the last 20 years [106, 247]. Since this thesis is primarily concerned with the collection and structuring of existing research, it makes sense that this is a major category of ITE Best Practices.

8.2. Algorithmic Detection of Violations: Bug Reports

In this section, I implemented 12 algorithms that detect violations to ITE Best Practices for Bug Reports. These Best Practices are specifically designed for Bug Reports, whether for data structure reasons (e.g., custom priorities on Bug Reports), or process reasons (e.g., the inherent time pressure that exists for high-priority Bug Reports). I discuss each of these algorithms in detail in the following subsections. Figure 8.1 is a summary figure of all findings across eleven of the ITE Best Practices for Bug Reports.[1] Each row represents a single Best Practice, where the violation detection algorithm was applied to the Bug Reports across the 13 Jira repos.

These "Bug Report" algorithms *can* also be applied to all other issue types, but the results should be interpreted with care. They can be applied due to the shared nature of the underlying issue-type data models, but there are differences in which values are used for certain fields (such as priority). Additionally, the recommendations of the ITE Best Practice were designed specifically to apply to Bug Reports. The result of applying these algorithms to other issue types can easily provide a false representation, or even a misleading one. For comparison purposes, I have applied these algorithms across all issue type themes (Activities), but I only discuss some of those results in this chapter (the results with meaningful and correctly interpreted outcomes), and the figures are in Appendix B to clearly separate and de-emphasise them.

Each row in Figure 8.1 is presented and discussed in greater detail in the following subsections. Figure 8.1 provides an overview of the detailed discussions below, and as the only place these results can all be seen next to each other. Six of the eleven sub-figures have their y-axis in symlog scale because the magnitude of those violation percentages are considerably low: generally between 1% and 10%. The other five, however, are presented with a traditional linear scale. This already says a lot about the differences across those violations, as some are much

[1] The twelfth algorithm was implemented outside the structure of this chapter, and therefore such a comparative figure cannot be produced.

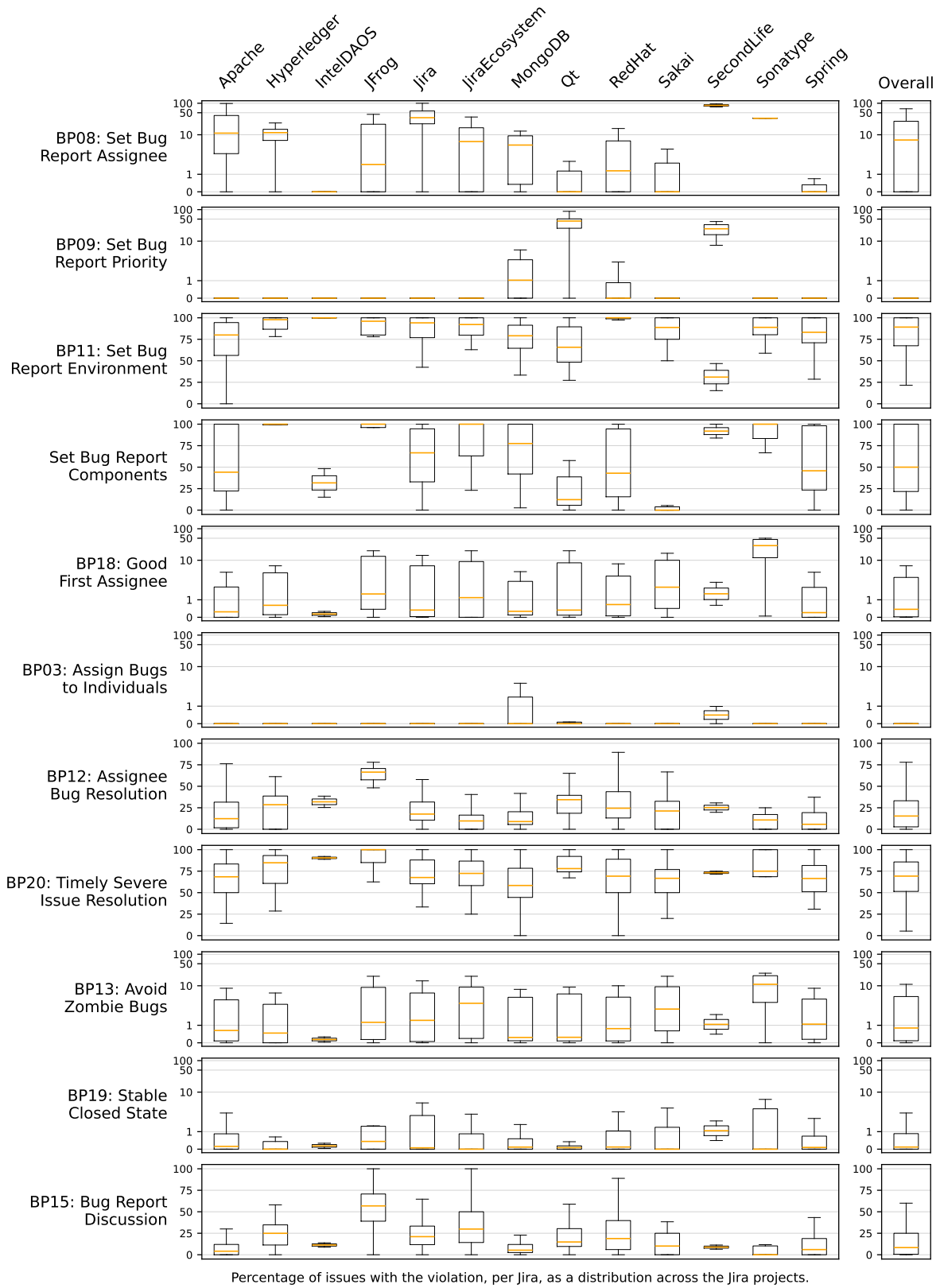


Figure 8.1.: Violations of Best Practices for Bug Reports.

more prevalent than others. Across the Best Practices, there are those with a median of 0% violations (considering all projects from all Jiras), some at 50%, and one at ~90%. This shows the diversity of Best Practice violations, in terms of "conformance" or "applicability" to this generalised set of Jiras.

8.2.1. Set Bug Report Fields

There is a group of Best Practices that are all related to setting Bug Report fields, including assignee (BP08), priority (BP09), severity (BP10), environment (BP11), and components.[2] These Best Practices, called "Set Bug Report [Field]" where [Field] is one of the above five fields, are all proposed by Qamar et al. [187, 188], and *assignee* is also discussed by Prediger [184]. These Best Practices are designed to encourage good Bug Report quality by having complete information on Bug Reports. Without these fields, it is not possible to trace aspects such as who worked on Bug Reports (assignee), how important it was to the organisation (priority), how critical the problem itself was (severity), and where the problem was applicable (environment).

Qamar et al. describe an algorithmic approach in which the bug must be both *fixed* (resolved) and *closed*, and yet the fields are still unassigned (empty). To capture the diverse ways in which Jira is used by different organisations, I performed an analysis of all potential statuses and resolutions, to produce a synonym list for what is "fixed" and what is "closed". Using the results of that analysis, I implemented these algorithms, as described by Qamar et al. [188].

> "First, we check whether the bug is fixed and closed. If so, we check whether the assignee field is empty or not. However, there are also some cases where the assignee field is not empty but an invalid email address is written. For instance, if the unassigned@gcc.gnu.org address is used as an assignee email, we consider this case a smell as well." [188]

I present the results of these algorithmic violation detections in Figures 8.2 (assignee), 8.3 (priority), 8.4 (environment), and 8.5 (components). Notably missing from these implementations is *severity*, which doesn't exist as a field in Jira.[3]

First, I discuss the "assignee" field, displayed in Figure 8.2. Overall, ~9% of Bug Reports suffer from this violation. Given the certainty of the detection method of this violation, this 9% is rather concerning. A fixed Bug Report must have been fixed by someone, and a closed state means that there is nothing more to do with this issue. Therefore, with near-certainty, these 9% of Bug Reports are missing information that should be there. This 9%, however, is not consistent across the Jiras. Some Jiras have as much as 30% of their Bug Reports missing the assignee (such as "Jira"), and others have a 0% violation rate, such as IntelDAOS. Future

[2] The "components" version is not in my Best Practice catalogue (due to no one formally proposing it in the literature), but I have the data to apply these algorithms to it.

[3] Online discussions regarding the severity field speculate that it is missing because it conceptually overlaps with *priority* too much, creating more confusion than it helps [168–170].

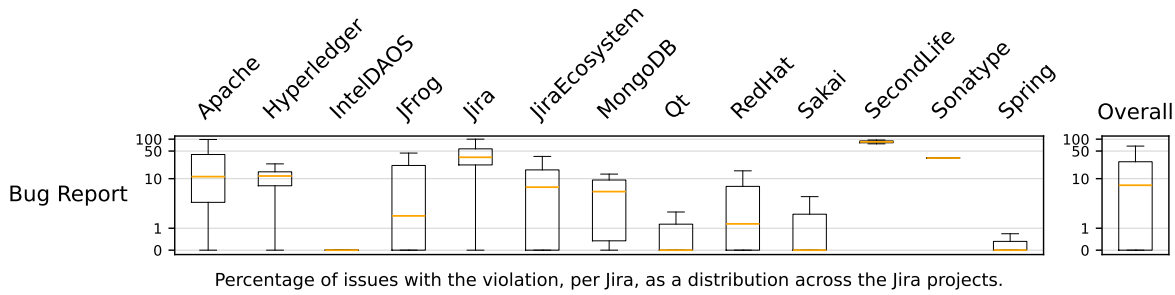


Figure 8.2.: Violations to Set Bug Report Assignee (BP08).

research should investigate the rationale behind excluding the assignee from Bug Reports in organisations with a high violation rate.

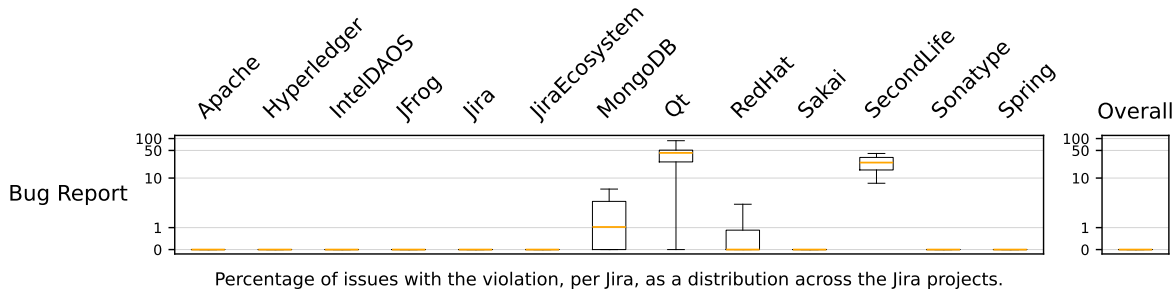


Figure 8.3.: Violations to Set Bug Report Priority (BP09).

Next, I discuss the "priority" field, displayed in Figure 8.3. Across the Jiras, there is a very low violation rate, i.e., very few Bug Reports are missing the priority. For the nine Jiras that have absolute 0% violations, this field could be mandatory on Bug Reports (or all issue types).[4] For the remaining four Jiras, it seems MongoDB and RedHat (with less than 10% violations) likely care about setting the priority, but for some reason do not enforce it, while Qt and SecondLife (with ~40–50% violations) do not use this field consistently.

Next, I discuss the "environment" field, displayed in Figure 8.4. This Best Practice has the highest violation rate across all Bug Report Best Practices, with a median project violation rate of ~90% across all Jiras. This violation rate is fairly similar across all Jiras, including some that have a median of 100% (they do not use this field at all), and some that are closer to 75% (barely use this field). My interpretation of these results is that setting the environment field on Bug Reports is not a priority.[5] An extension of this interpretation is that setting the environment field on Bug Reports is not a Best Practice. However, it should also be considered that perhaps there are good reasons to have this Best Practice. Future research needs to provide

[4]Looking at the complete results across all issue types, presented in Figure B.2, it indeed seems that "priority" is a required field across most issue types.

[5]The violation rate on Requirements and Development issue types are even higher (see Fig. B.3).

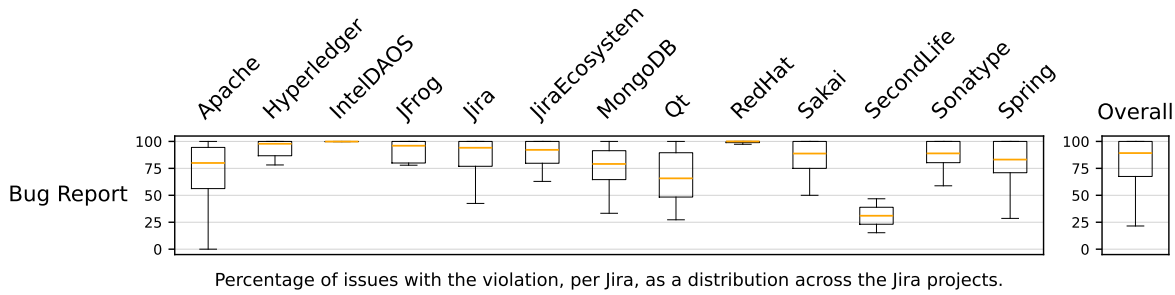


Figure 8.4.: Violations to Set Bug Report Environment (BP11).

strong empirical evidence to the claim that setting the environment field on Bug Reports is indeed an ITE Best Practice that should be followed.

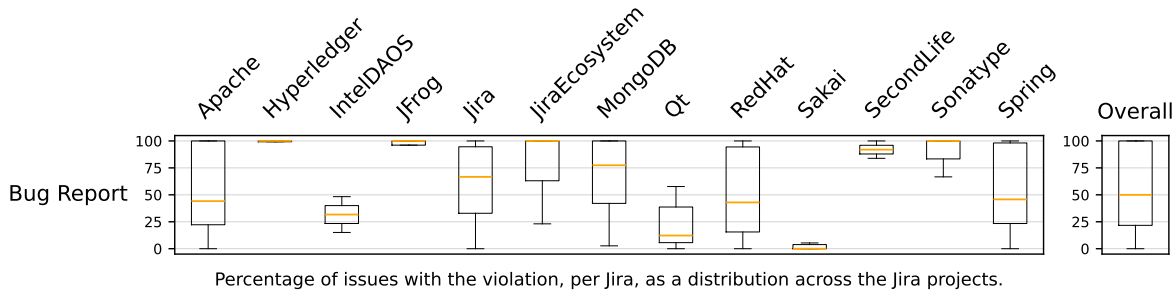


Figure 8.5.: Violations to Set Bug Report Components (no catalogue item).

Next, I discuss the "components" field, displayed in Figure 8.5. The results show a varying degree of violations, from 100% (Hyperledger), to near 0% (Sakai). The median across all projects is 50%, which highlights the varying degree of violations. It seems this Best Practice is important in some contexts, and not in others. Further evidence for this context-specific assumption is the wide variance on the upper- and lower-quartiles, across multiple Jiras. The evidence suggests that whether projects prioritise the environment field on Bug Reports is very project-dependent, with many at 100% violation rate (they do not consider their absence to be a violation), and many near 0% (they heavily use this field). Apache, RedHat, and Spring, for example, have upper-quartiles near 100%, and lower-quartiles near ~25%. If an organisation wants to implement the Set Bug Report Components Best Practice, they should allow for project-specific opt-out to satisfy the needs of diverse project types.

As described in previous Best Practices, I also visualised the results for all issue type themes in figures available in the Appendix B. Interestingly, there is little difference across the issue type themes, compared to that of Bug Reports. There is a near-identical trend in the percentage of violations, per Jira, across the issue type themes. This trend holds largely true for all "Set Bug Report Fields" Best Practices, and across all organisations in my dataset.

8.2.2. Good First Assignee

Good First Assignee (BP18)

The "Good First Assignee" Best Practice (see Table A.19) was proposed and later extended by Qamar et al. [187, 188]. The Best Practice is designed to decrease bug resolution time by ensuring a good first assignee on Bug Reports. The idea being that performing multiple re-assignments increases the time to complete bugs [187, 188]. The algorithmic approach is to look for evolutions of the assignee field, since every evolution is a re-assignment. I don't count the "creational" evolutions, where the assignee is first set. Qamar et al. also recommend not to count rapid reassignments: when an assignee is re-assigned within a 5-minute interval.

> "The mining strategy for this smell is to look in the bug history for the assignee property. If the assignee field is changed more than twice, then we count it as a smell. Also, we observed that there are some multiple assignee changes done in a very short interval. We consider these multiple assignee changes as a mistake and do not count them as a smell. The interval duration is set as five minutes. Therefore, if multiple assignments are done in five minutes, they are counted as one assignment." [188]

While Qamar et al. count violations as "more than twice" (without justification), I count violations as more than once. The first assignment is necessary, but the second assignment is already the first re-assignment. I present the results of this implementation in Figure 8.6.

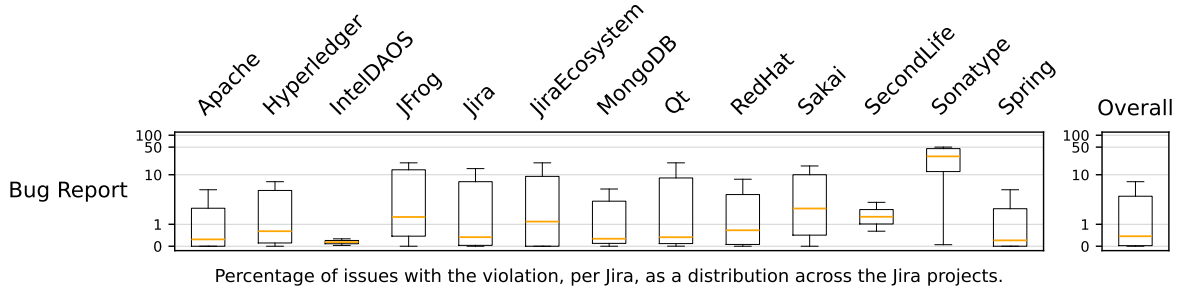


Figure 8.6.: Violations to Good First Assignee (BP18).

The results show that this violation is quite rare, with a median occurrence rate of ~0.5% This highlights the importance of this Best Practice to these organisations. The log-scale figure shows that most Jira Repos are at around 1% violation rate, with only Sonatype having a high rate at ~30%. Future work can unpack why Sonatype's violation rate is so high.

Assign Bugs to Individuals (BP03)

The "Assign Bugs to Individuals" Best Practice (see Table A.4) was proposed and later extended by Qamar et al. [187, 188]. The Best Practice is designed to foster traceability and ownership

of Bug Reports by linking the individual assignee to the issue *before it is resolved* (closed). A not uncommon practice is to assign a team, group, or department to a Bug Report, which is then later assigned to one of the individuals within that team. However, it is sometimes the case that someone will fix the bug and resolve the Bug Report, without assigning themselves to the Bug Report. Without such a Best Practice, it becomes difficult to follow up on resolved bugs, particularly if they need to be reopened or investigated.

Qamar et al. describe a programmatic approach to detecting this smell [187, 188]. Search for Bug Reports that are resolved, and then check if the assignee is an individual. I implemented their approach and applied it to my Jira dataset, producing the results displayed in Figure 8.7.

> "First, we check whether the bug is assigned. If so, we search for the selected keywords: 'team', 'group' and 'backlog' in the assignee field. We found those keywords by manually inspecting the assignee names in each project. For example, in the MongoDB Core Server project history, there are some bugs assigned to Backlog-Sharding Team, which we consider it as a smell." [188]

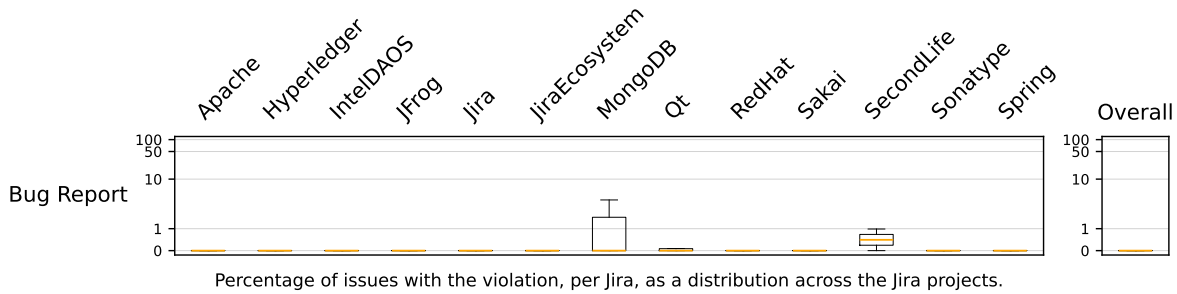


Figure 8.7.: Violations to Assign Bugs to Individuals (BP03).

There are very few violations to this Best Practice. MongoDB and SecondLife have the most violations. MongoDB has an upper quartile at ~2%, and an upper whisker at ~5%. SecondLife has the only median above 0%, at ~0.5%, and a tight upper and lower quartile only ~0.25% above and below. These results show that the Assign Bugs to Individuals Best Practice is rarely violated.

As described in the Best Practice table (Table A.4), this is a Best Practice that is only contextually relevant to Bug Reports. Figure B.6 also displays the results for the three issue type themes, Requirements, Development, and Maintenance. MongoDB and SecondLife are again the only two Jiras that show any notable violations, and they continue to be low across the issue type themes. Development is the only issue type that shows effectively no violations, which is an interesting area for future study.

Overall, the Assign Bugs to Individuals Best Practice is closely followed by the organisations under study. Whether implicitly or explicitly, organisations seem to agree that Bug Reports (and other issue types) should be assigned to an individual before they are finally resolved.

Assignee Bug Resolution (BP12)

The "Assignee Bug Resolution" Best Practice (see Table A.13) was proposed and later extended by Qamar et al. [187, 188]. The Best Practice is designed to foster ownership and traceability between the assignee of a Bug Report, and the process of resolving it. If anyone can resolve the report, then there is less ownership on the assignee to act on it. Qamar et al. describe a programmatic approach to detecting this smell [187, 188]. Search for Bug Reports that are resolved, and then check the assignee against the resolver. I implemented their approach, producing the following results displayed in Figure 8.8. Listed verbatim, their approach is:

> "First, we check whether the bug is assigned and resolved. If so, we compare the assignee and the person who resolved the bug." [188]

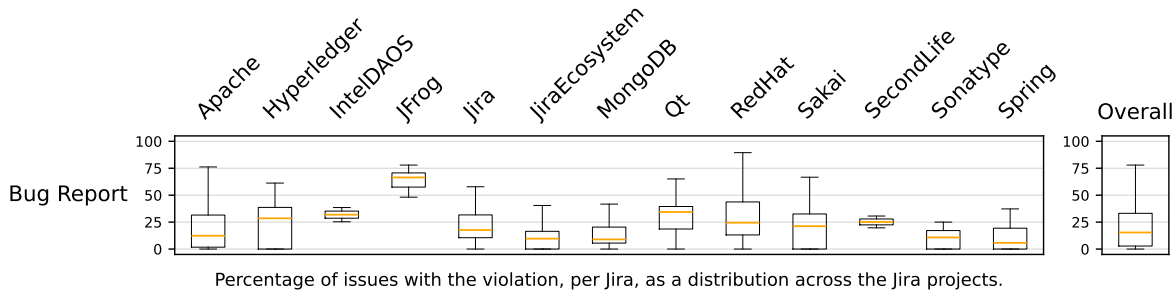


Figure 8.8.: Violations to Assignee Bug Resolution (BP12).

All Jira Repos have some number of violations to this Best Practice, from ~10% (Spring) all the way up to ~65% (JFrog). The IQRs are rather large, and the whiskers are also rather pronounced. These results show that most projects in my dataset do not conform to this Best Practice. Given the logical nature of this Best Practice, and the benefits provided, future work should investigate why this does not appear to be important within industrial projects.

8.2.3. Closing Bug Reports

Timely Severe Issue Resolution (BP20)

The "Timely Severe Issue Resolution" Best Practice (see Table A.21) was proposed by Halverson et al. [94] and Prediger [184]. The Best Practice is designed to encourage quick resolution of Bug Reports. In particular, quick resolution of Bug Reports that have been identified as "severe" in some way, either by the customer or the organisation. The basic idea is that when a Bug Report is labelled as "highly severe" (a definition which differs from organisation to organisation), then the Bug Report should be closed (and hopefully resolved) within 48 hours. The choice of 48 hours is simply stated as the default time chosen by one of the "informants" of Halverson et al. [94]. To detect violations to this Best Practice, we first have to identify what a "high severity" is within each Jira Repo. I performed a manual analysis of all possible Priority values

in my dataset, looking for those to label as "high" severity.[6] For each potential value, I have the name of the Priority, as well as how many issues it was used in. With these two pieces of information, here are the values I chose to mean "high severity":

- **($>$ 1,000 issues each)**: Critical, Blocker, P1: Critical, Highest, Critical - P2
- **($<$ 1,000 issues each)**: Urgent, Blocker - P1, P1, 2 - Critical, P1-Urgent, P0, 1 - Blocker
- **($<$ 100 issues each)**: P2-Critial, P1-Blocker, Blocking, Severe

I then compare the closed date to the created date. For my analysis, I selected a timeframe of 7 days. My familiarity with the customer support domain [161, 162, 166] has taught me that while 48 hours is commendable, it is not reasonable for many organisational contexts. Instead, I selected a more relaxed one-week timeframe, allowing time for organisational units to connect and resolve the issue. I present the findings of this algorithm in Figure 8.9.

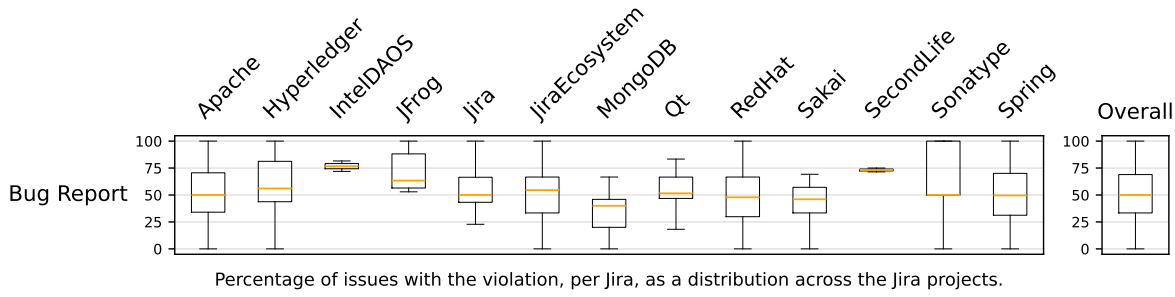


Figure 8.9.: Violations to Timely Severe Issue Resolution (BP20).

The results show that, despite the generous 7-day resolution window, a median of ~50% of issues across the projects violate this Best Practice. The 50% is fairly consistent across the Jira Repos, except IntelDAOS and SecondLife at ~75%. This result is difficult to interpret without diving deeper into the data, and cross-checking with stakeholders within these organisations. They might conform to the idea that high-priority Bug Reports should be solved quickly, but only meet this 7-day threshold 50% of the time. It could also be that response and resolution times in OSS projects are longer than in projects that normally cater to paying customers. Regardless, it is clear that even at 7 days, this Best Practice is often violated in my dataset.

Avoid Zombie Bugs (BP13) & Active Bug Reports (BP14)

The "Avoid Zombie Bugs" Best Practice (see Table A.14) was proposed by Halverson et al. [94], and the "Active Bug Reports" Best Practice (see Table A.15) was proposed by Qamar et al. [187, 188] and Aranda and Venolia [10]. These Best Practices were designed to foster a meaningful backlog by addressing Bug Reports before they get too old. The idea being that after a certain

[6] Jira does not have a "Severity" field; rather, they use the Priority field.

period of inactivity, Bug Reports must either be closed, or worked on. Of course, if organisations had the time to address all their Bug Reports, they would, so the problem is clearly resources, not desire. However, this Best Practice emphasises a *meaningful* backlog, which means that if something is in the backlog, it must still have meaning to the software. Not all closed Bug Reports need to be resolved, and this Best Practice encourages closing Bug Reports that the organisation has no intention of addressing *in the near future*. How to define "near future" is the hard part. Neither Halverson et al. [94] nor Aranda and Venolia [10] specify a timeline, whereas Qamar et al. [188] specify a timeline of three months. To detect violations to this Best Practice, I review all evolutions to every issue, and count the issues that have a gap in activity of three months or more. I present the findings of this algorithm in Figure 8.10.

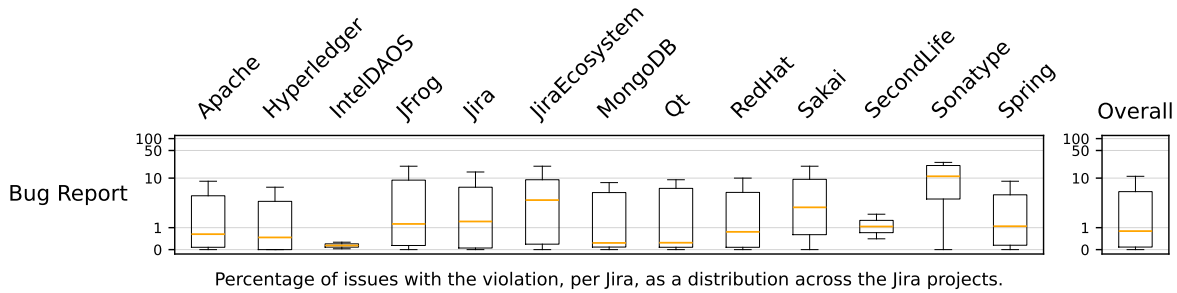


Figure 8.10.: Violations to Avoid Zombie Bugs (BP13) & Active Bug Reports (BP14).

Across the organisations and projects, there is a median violation rate of ~1%. This shows that practitioners are interested in keeping Bug Reports active and therefore meaningful. Sonatype is an outlier, with a median violation rate of 10%; however, this is still relatively small, and shows an overall commitment to active Bug Reports. MongoDB and Qt show a particular commitment to this Best Practice, with a median at almost 0%.

Stable Closed State (BP19)

The "Stable Closed State" Best Practice (see Table A.20) was proposed by Qamar et al. [187, 188]. The Best Practice is designed to improve stability of issues by discouraging the repeated opening and closing of Bug Reports. Repeated opening and closing of bugs decreases software quality, increases maintenance costs, and is frustrating for developers [188]. To detect these violations, I review all evolutions to the Status field, looking for issues that were re-opened. In alignment with the recommendations from Qamar et al. [188], I do not count "rapid" cycles that occur in less than 5 minutes. Any change in the Status that occurs less than 5 minutes before the last Status change is ignored. I present the findings of this algorithm in Figure 8.11.

> "We check the history of the status field of the bug. Some projects explicitly use REOPENED value for the status field while others do not. In such cases, we check whether the status field is changed from Closed to another value, and we count it

as a smell. Also, we observed that there are some multiple status changes in a very short interval. We consider these changes a mistake and do not count them as a smell. The interval duration is set as five minutes. Therefore, if a bug status is changed multiple times in five minutes, it is counted as one change." [188]

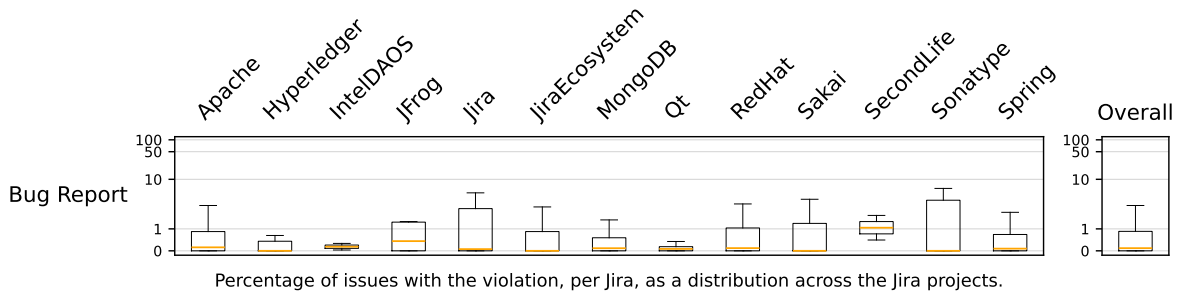


Figure 8.11.: Violations to Stable Closed State (BP19).

As shown by the results, violations to this Best Practice are extremely rare, with a median violation rate of ~0.01%. This is a clear sign that these organisations follow this Best Practice. There is some variance in the IQRs and whiskers, but even then, the average is upper quartile is around 1%. Jira (the organisation) and Sonatype have the most violations to this Best Practice, with upper whiskers at ~8%. Given the low medians across the Jira Repos, it is likely they would prefer to avoid these violations altogether, and therefore would greatly benefit from processes or automations that support adherence to this Best Practice.

8.2.4. Bug Report Assorted

Bug Report Discussion (BP15)

The "Bug Report Discussion" Best Practice (see Table A.16) was proposed by Qamar et al. [187, 188] and Tamburri et al. [218]. The Best Practice is designed to foster activity on Bug Reports through discussions in the comments. The idea is that the comments on Bug Reports are part of a healthy OSS community, and lack of comments is a sign that something is wrong. To detect this violation, I review all closed Bug Reports, and check for those with no comments. I present the findings of this algorithm in Figure 8.12.

> "First, we check whether the bug is closed. If so, we check whether there is at least one comment in the bug." [188]

The results overall show a median violation rate of ~10% across the Jira Repos. However, there is a mix of Jira Repos with almost no violations (e.g., Sonatype with 0%, and MongoDB with ~5%), and others such as JFrog with almost 60% violation rate. One thing to consider is the bots that some projects have installed, which may be offsetting their violations. Another thing to consider is the popularity of the projects, and how that affects the outside discussion on

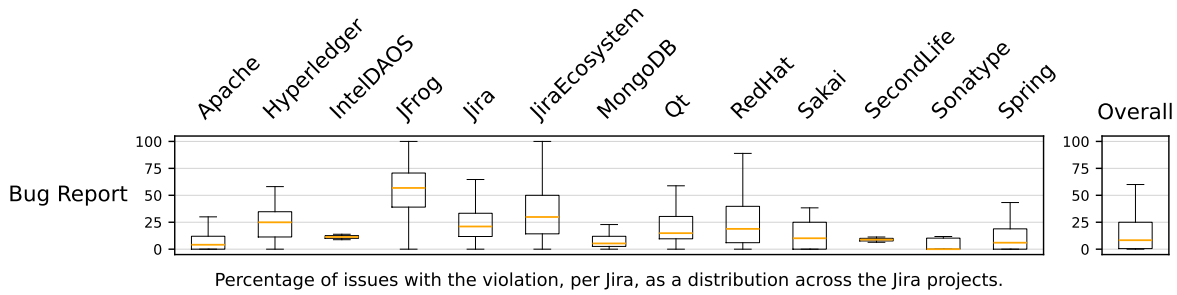


Figure 8.12.: Violations to Bug Report Discussion (BP15).

their Bug Reports. For a full and in-depth analysis, one could dig into whether the comments were bots, outsiders, or maintainers.

8.3. Algorithmic Detection Violations: All Issue Types

In this section, I implement 6 algorithms that detect violations to ITE Best Practices for all issue types. I discuss each of these algorithms in the following subsections. Figure 8.13 is a summary of all findings across the 6 algorithms. Each row represents a single Best Practice, where the violation detection algorithm was applied to all issues across the 13 Jira Repos.

Five of the six sub-figures have their y-axis in symlog scale—highlighting the rarity of these violations, while only one is in linear scale. Unlike the Bug Report Best Practices discussed presented above, these "all issue types" Best Practices have some trends across Jiras. For example, Summary Length and Avoid Status Ping Pong have roughly the same violation rate across the Jiras (50% and 10%, respectively). Unlike the Bug Report Best Practices in Figure 8.1, most of the Best Practices have quite low violation rates, except for Summary Length. The following subsections go into detail regarding each of these Best Practices.

8.3.1. Summary and Description Length

Sufficient Description (BP04)

The "Sufficient Description" Best Practice (see Table A.4) was proposed by Prediger [184]. The Best Practice is designed to ensure that issue descriptions are up to a certain standard when it comes to length and basic information. It is common to see issues with just a title and no description, or, if a description exists, it is just a short sentence. This is likely due to the tacit knowledge the reporter assumes is known to everyone, but is likely not.

Prediger does not describe specifically how to detect this violation, although she discusses the concept of length (compared to that of, for example, semantic content). Prediger does not specify the exact length that she considers "sufficient", likely because there is not a straightforward answer that applies universally to all issue types across all organisations. That is a

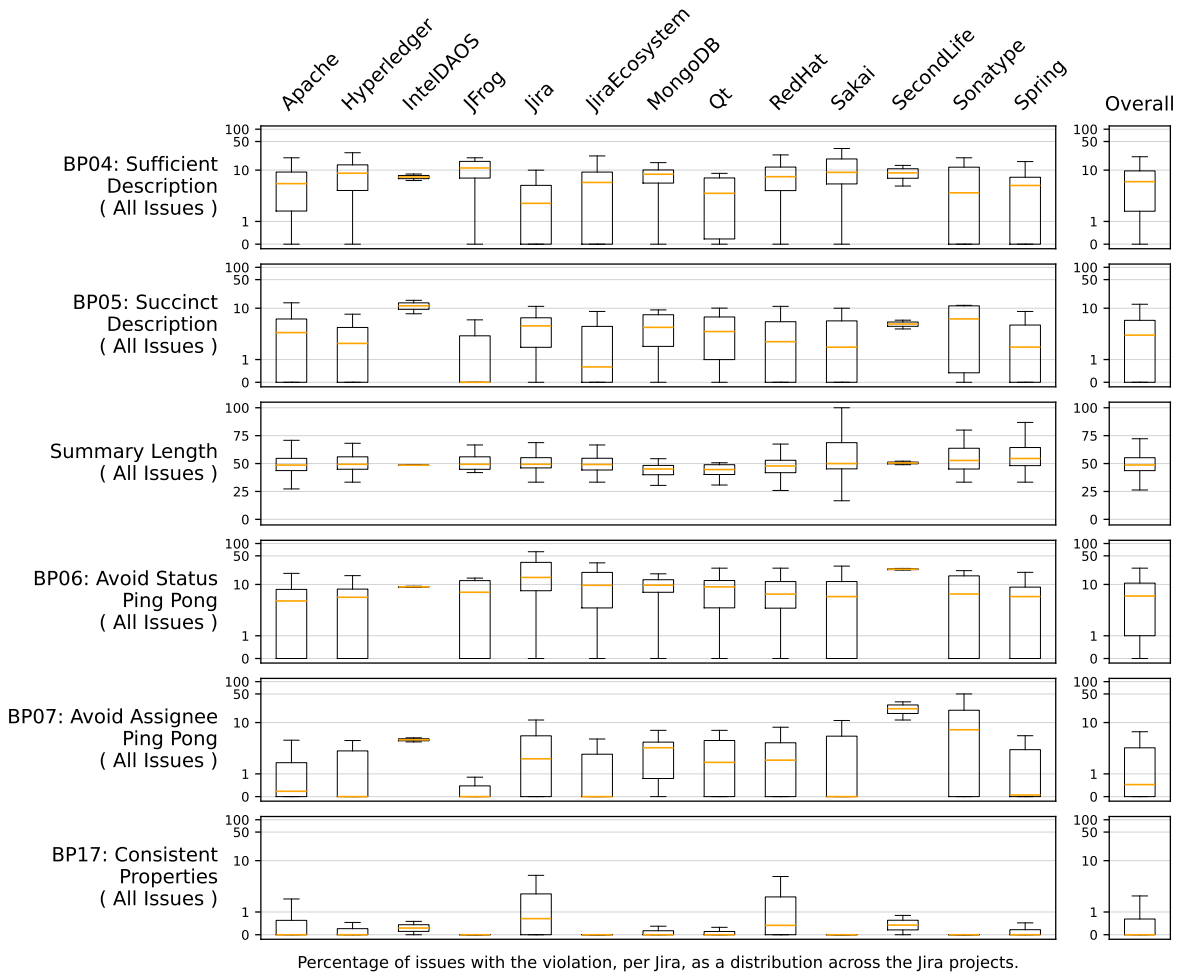


Figure 8.13.: Violations of Best Practices for all issue types.

sign that this Best Practice needs much more research to understand the contextual factors that influence the application and tuning of this Best Practice. Regardless, I implemented and applied an algorithm that automatically detects violations of this Best Practice, with a tunable threshold of length. For the initial threshold, I set it to 10 words. I think it is easy to argue that 10 words is not sufficiently long enough for a description to adequately describe an issue. However, for the purpose of tuning a violation detection algorithm, I chose a number that is not likely to produce many false positives. In other words, detected violations are indeed violations, thus not wasting the time of stakeholders who are being notified of this violation. I present the results of this algorithmic violation detection in Figure 8.14.

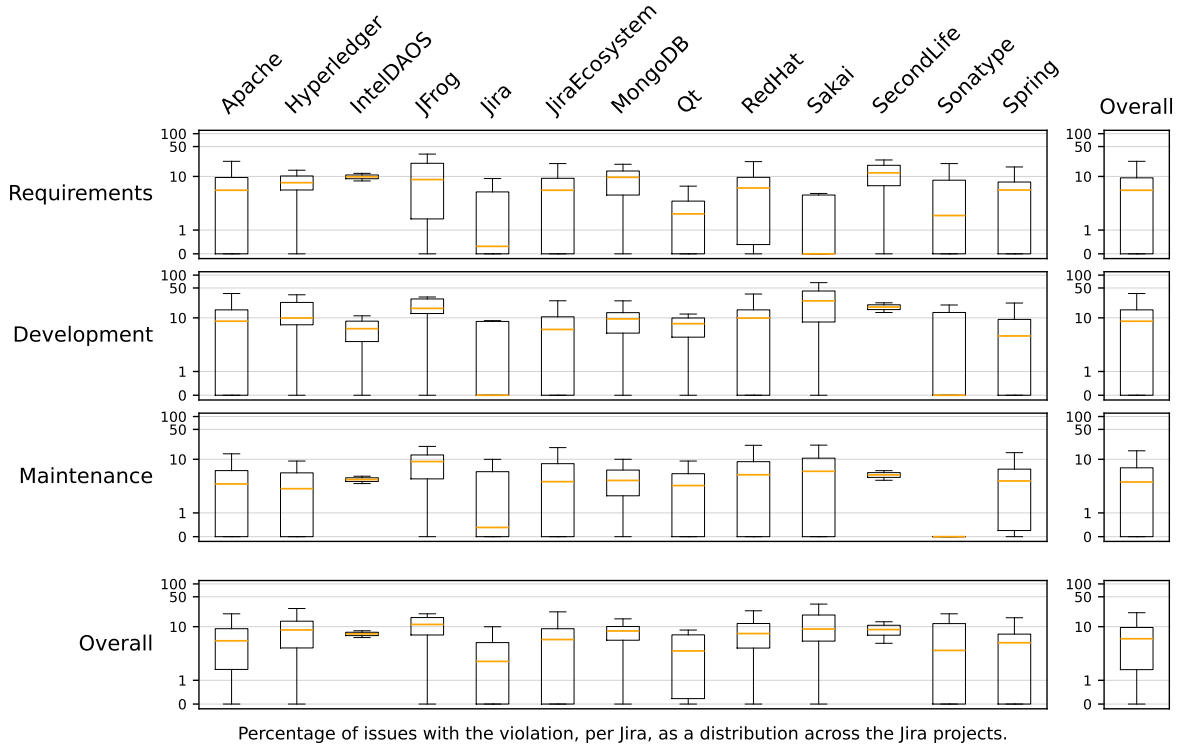


Figure 8.14.: Violations to Sufficient Description (BP04).

Despite the generously low threshold of 10 words, most Jiras across all issue types suffer from 5–10% violations per project (median). Interestingly, Development issues appear to be struck the most by this violation (at ~10% median), although the difference between Requirements and Maintenance is only a few percentage points. It is also worth noting that some of the Jiras have lower quartiles at 0%, which means there is a substantial number of projects (~25%) that do not suffer from this violation at all.

Succinct Description (BP05)

The "Succinct Description" Best Practice (see Table A.6) was proposed by Lüders [137]. The Best Practice is designed to ensure that issue descriptions are not too long, which adds unnecessary wasted time and potential misunderstandings from misinterpretations of the text. While issue descriptions should be sufficiently long ("Sufficient Description" Best Practice Table 8.14), this does not mean that longer is universally better. As with many variable concepts, there is a trade-off with a meaningful compromise somewhere in the middle.

Lüders does not describe specifically what threshold counts as "succinct" versus not, which is similar to the threshold of Sufficient Description. While it may not be possible to pick a universal threshold, we simply do not know without further research investigating the contextual factors that influence this Best Practice. Regardless, I implemented the detection of this violation, and set the initial threshold at 250 words. I chose this threshold because that is the standard length of a page in a book, and represents multiple paragraphs of text. At this point, it is unlikely that additional words are going to make the description any clearer, and perhaps editing is necessary if additional information is required. I present the results of this algorithmic violation detection in Figure 8.15.

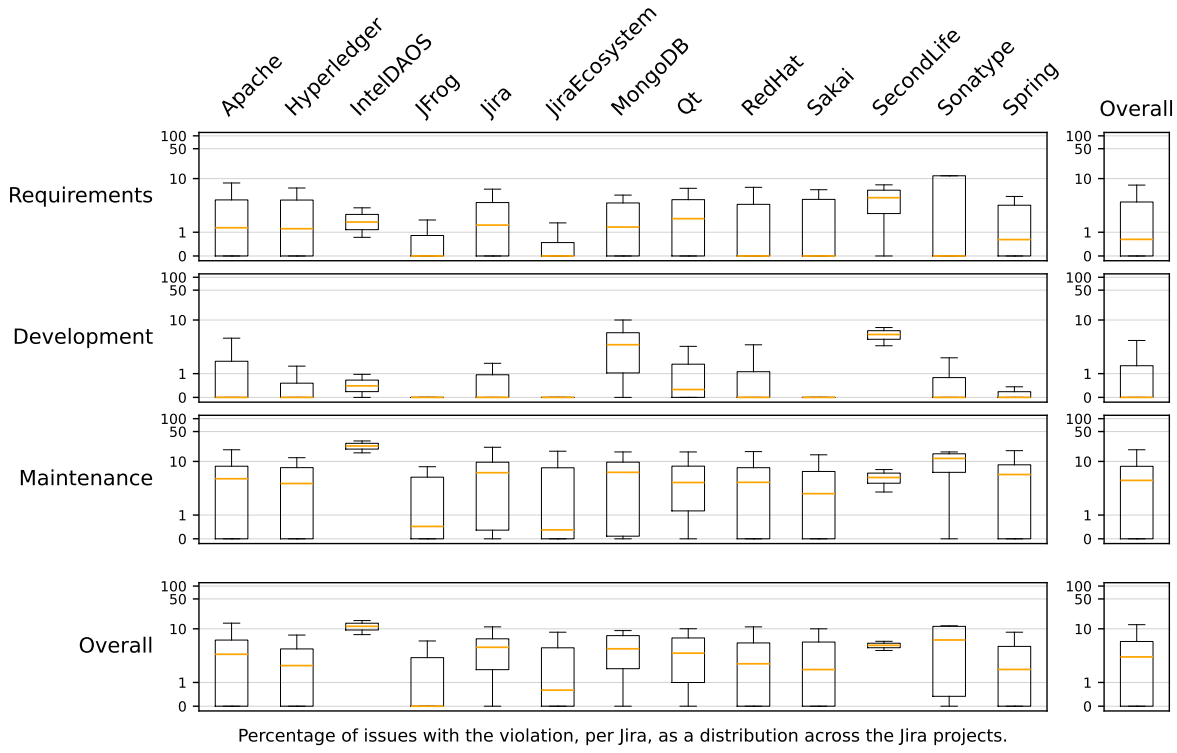


Figure 8.15.: Violations to Succinct Description (BP05).

The results show an interesting divide between Requirements, Development, and Maintenance issues: Development issues rarely suffer from this violation (~0% median), Requirements

minimally suffer from this violation (~1% median), and Maintenance suffers from this violation much more (~8% median). This suggests that either Maintenance issues are of lower quality when it comes to succinct descriptions, or, Maintenance issues have a necessity for more information, and therefore longer descriptions. Both of these potential phenomena require additional research to understand better. For the purpose of this Best Practice, these results suggest that different issue types require different thresholds for what is considered a "succinct description". Further research could uncover and prescribe what these thresholds are.

Summary Length

The "Summary Length" Best Practice is not listed in the Best Practice catalogue, since it has not been formally proposed within the literature. The reason for including it in this algorithms section is the transferability of the algorithms written for the Description field. The Summary and Description field both share a similar role when it comes to describing the issue. Perhaps more important for the Summary, however, is the exact length of the text. While the Description can vary quite drastically depending on the type of issue being described, the Summary should always be a consistent length. What defines "consistent" for a given project is not so clear. Bohn describes an ideal range of 39 to 70 characters in his dissection of Summary length, citing literature [250], company guidelines,[7] and his own personal investigations [29]. Since the focus of my work is on implementation and application, and not the empirical investigation of what is the ideal length, this range will suffice. Using this range of 39–70, I detect and flag all summaries that are outside this range as violations. I present the results of this algorithmic violation detection in Figure 8.16.

With the current threshold, ~50% of issues across the projects are violating this Best Practice. This is fairly consistent across the Jira Repos and activity types, with varying degrees of variance visible in the IQRs. Given the importance of the Summary field, and the emphasis on it as a central way to communicate and identify the issue, I don't think this ~50% violation rate is representative of the real impact of this Best Practice. This number should be much lower. While I believe keeping the Summary within a certain range is important, and likely indirectly enforced within many projects, I don't think it is as easy as a single threshold you can apply universally across projects. Context likely plays an important role here, and without it, these results are lacking in realism.

[7] Bug Writing Guidelines by Bugzilla (`bugzilla.mozilla.org`) and eclipse (`bugs.eclipse.org`).

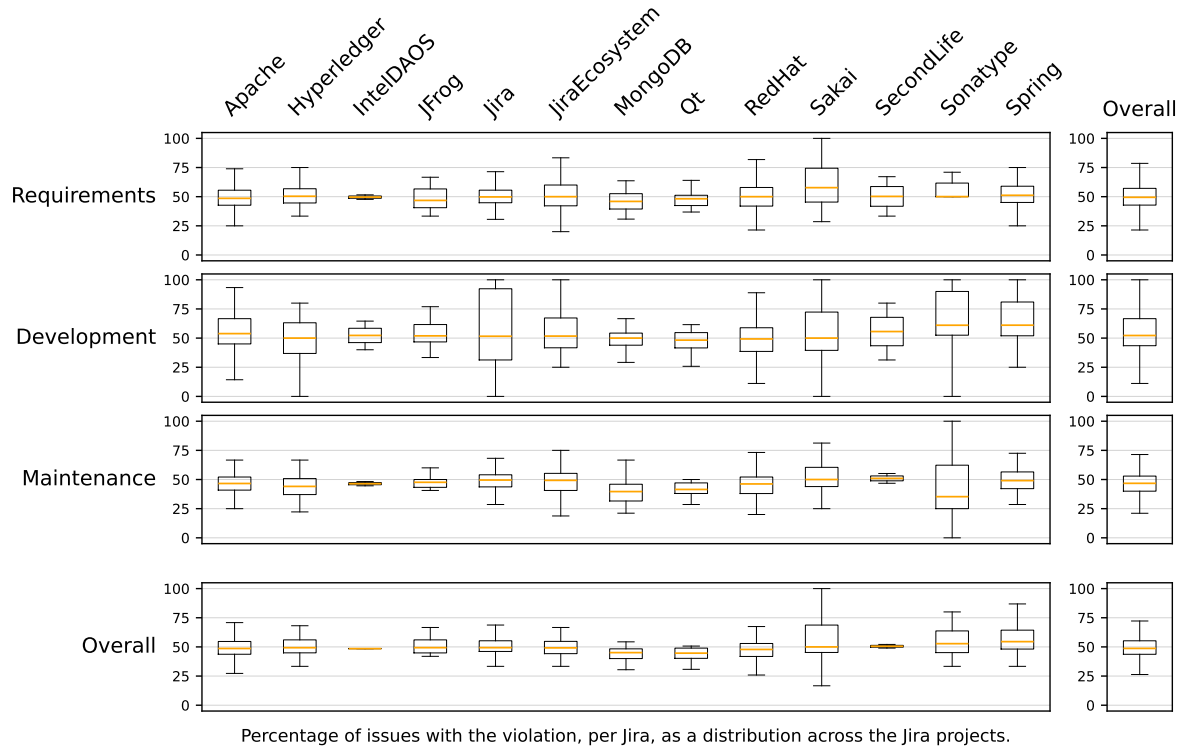


Figure 8.16.: Violations to Summary Length.

8.3.2. Stable and Correct Fields

Avoid Status Ping Pong (BP06)

The "Avoid Status Ping Pong" Best Practice (see Table A.7) was proposed by Halverson et al. [94], Aranda and Venolia [10], and Prediger [184]. The Best Practice is designed to ensure that issues go through linear, intentional *status* changes from open to closed. If not adhering to this Best Practice, cycles can form within the status, which likely represents duplicate work and confused stakeholders. To detect this Best Practice, create a state-transition graph from the history of all status changes in an ITS, and search those graphs for cycles. If an organisation has "allowed" cycles (such as between "QA" and "Dev"), then add these as exceptions within the algorithms. I present the results of this algorithmic violation detection in Figure 8.17.

The results show that status cycles are actually quite common across the projects. The median is ~10% for all Jira repositories, although repos such as IntelDAOS and SecondLife show medians between 25 and 50%. Given the prevalence across the Jira repos, this suggests that there are likely well-known and accepted cycles that occur. To further refine this violation detection to remove what are likely false-positives, an analysis of the existing cycles needs to be done—per repo. Then, once the researcher has a suitable understanding of *what* is going on, they then need to approach the stakeholders within these repositories to understand *why*

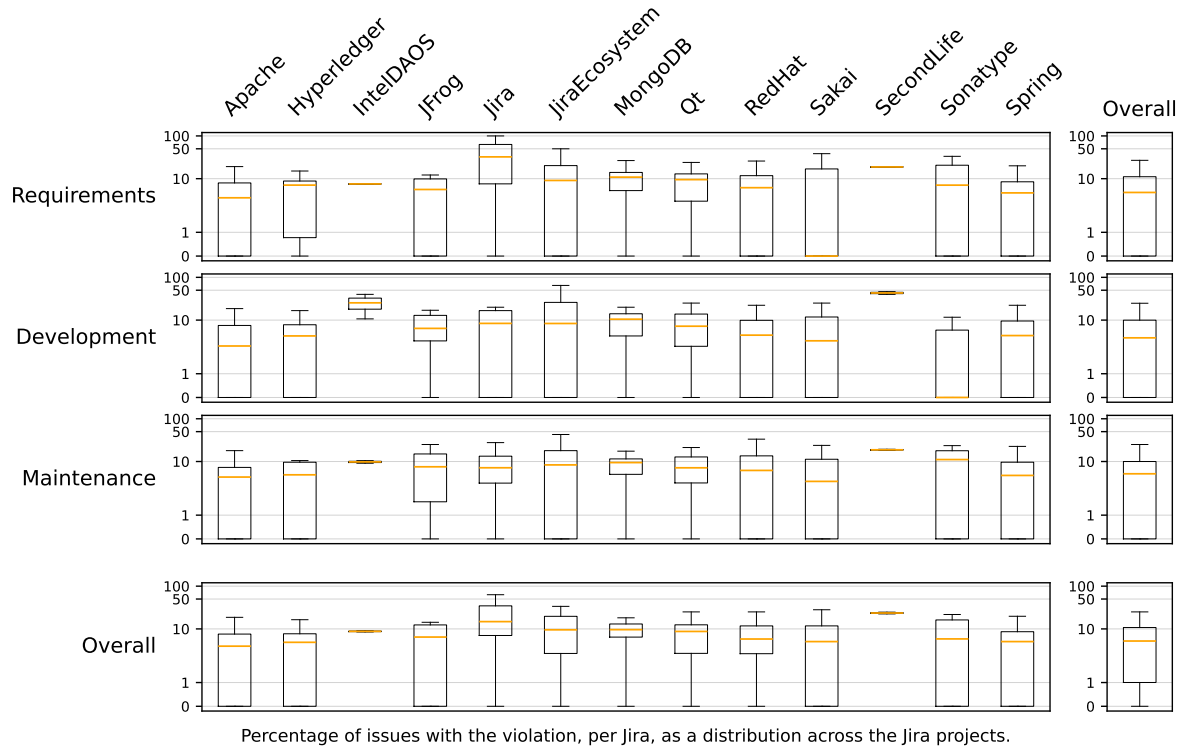


Figure 8.17.: Violations to Avoid Status Ping Pong (BP06).

these cycles are so common. Some may be unintentional, but the more common and cross-repo cycle patterns are likely accepted forms of process for these issues.

Avoid Assignee Ping Pong (BP07)

The "Avoid Assignee Ping Pong" Best Practice (see Table A.8) was proposed by Halverson et al. [94], Aranda and Venolia [10], and Prediger [184]. Similar to "Avoid Status Ping Pong" above, this Best Practice is designed to ensure that issues go through linear, intentional *assignee* changes. When the assignee is re-assigned back and forth between different people, this is a sign that there is a disagreement in some external process or responsibility. This can only be resolved through discussions regarding who should actually be working on it. To detect this Best Practice, create a state-transition graph from the history of all assignee changes in an ITS, and search those graphs for cycles. If an organisation has "allowed" assignee cycles, such as between people co-working on the issue, then add these as exceptions within the algorithms. I present the results of this algorithmic violation detection in Figure 8.18.

The results show that assignee cycles are quite rare, with an overall median of ~0.5%; however, roughly half of the repos have an assignee cycle median between ~5–10%. The SecondLife (~25%) repo has a high count of assignee cycles.

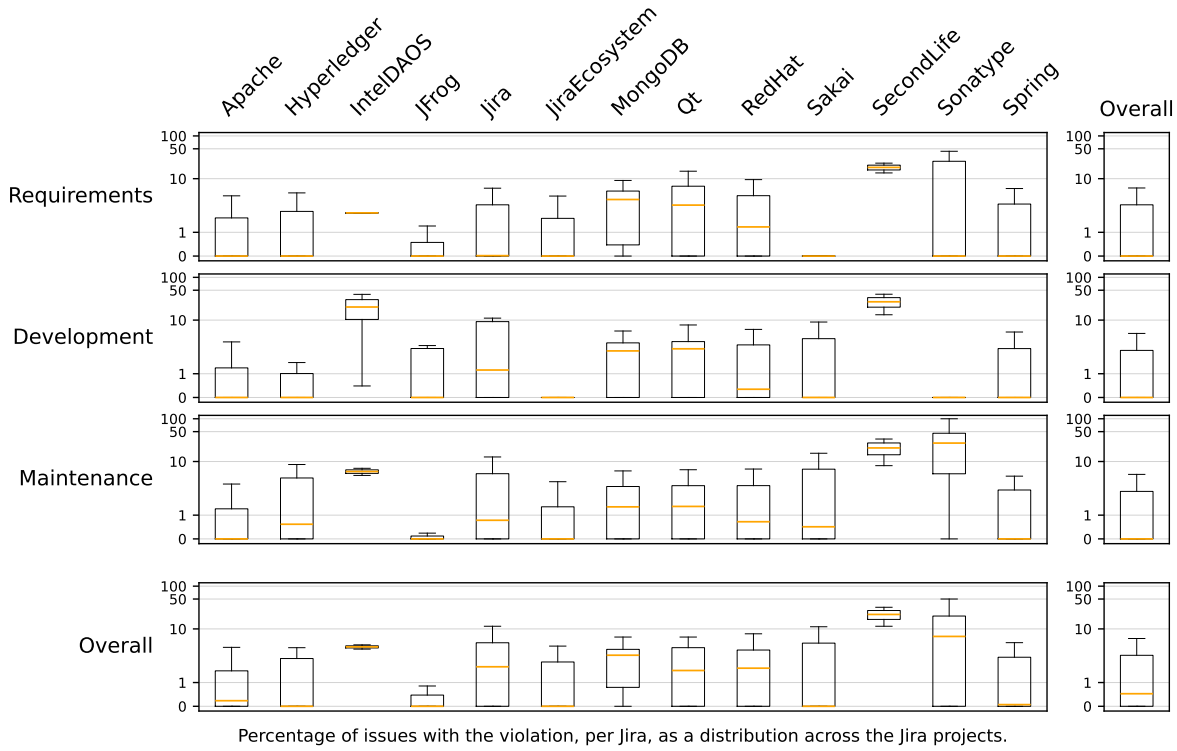


Figure 8.18.: Violations to Avoid Assignee Ping Pong (BP07).

Consistent Properties (BP17)

The "Consistent Properties" Best Practice (see Table A.18) was proposed by Lüders [137]. The Best Practice is designed to keep the fields within issues as up-to-date as possible, considering information that is sometimes written in the Description or Comments fields. Despite fields that are designed to hold specific information, stakeholders sometimes write things such as "this issue is now resolved", without setting the "Resolution" field. This Best Practice states that no one should state information else where on the issue when the fields could instead be updated.

Lüders does not propose a specific algorithmic approach to identifying violations of this Best Practice. Given that these violations occur in natural language (in the Description or Comments fields), detection becomes an NLP problem. I chose to develop a preliminary, naive implementation of this detection algorithm using context-dependent information and text searching. First, I collect a list of all fields that exist within issues (which I already conducted in Chapter 5). Then, I search the full evolution history of all issues for values those fields were ever set to, separated per Jira. Finally, I search the Description and Comments for any instance of both a field,[8] and one of its possible values (within the context of the Jira). Like most NLP

[8]For computation reasons, I only search for the fields IssueType, Status, Priority, and Resolution. These fields have historical values in the range of <200, whereas the other fields have upwards of 10,000 potential values for each field, per Jira. Future research can investigate why this is, and how to reduce the computational complexity to make this naive approach feasible for these fields.

tasks, the results are fraught with false positives, but the recall will be nearly 100%, except in cases of spelling mistakes. As Dan Berry describes, recall is often preferred over precision, since a human can quickly assess whether a result is correct or not, but finding missed results requires an extraordinary amount of time [25]. Accordingly, I left my preliminary implementation as is. I present the results of this algorithmic violation detection in Figure 8.19.

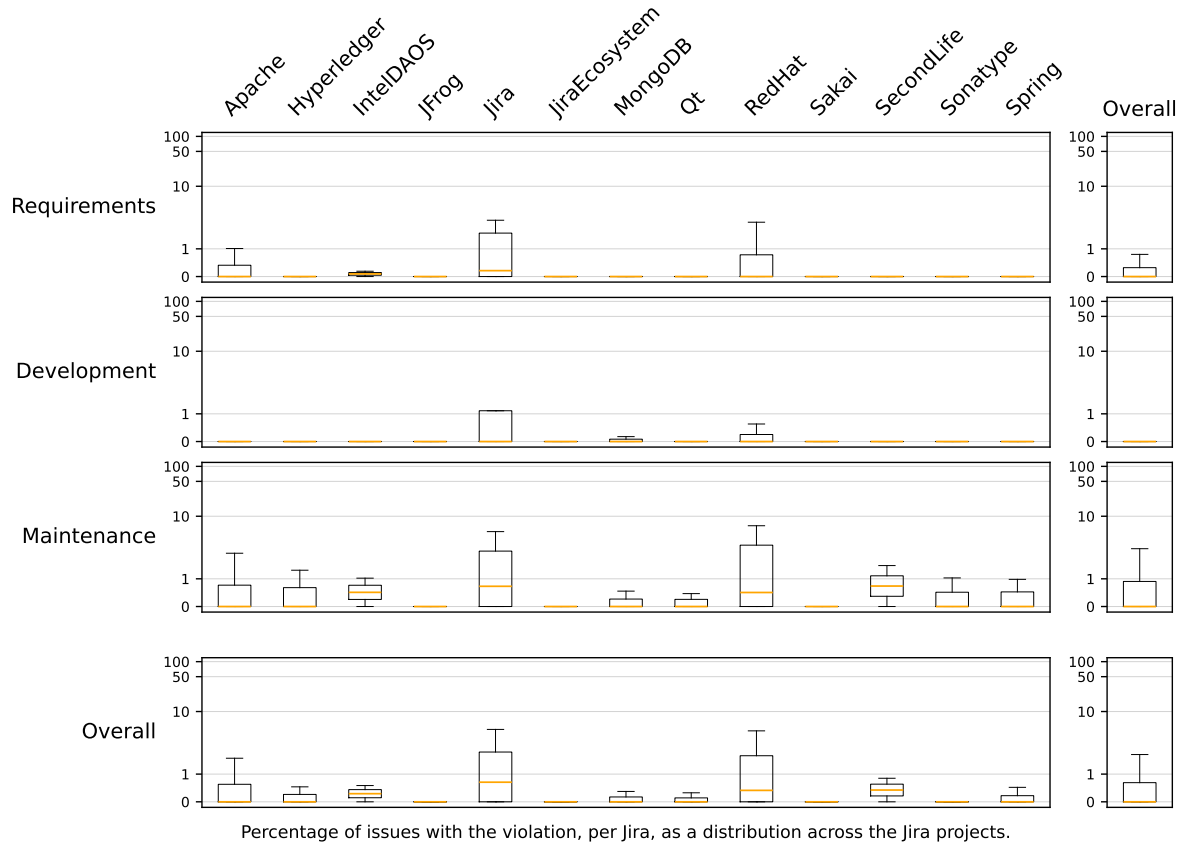


Figure 8.19.: Violations to Consistent Properties (BP17).

There are almost no violations to this Best Practice, across all Jiras, despite the admittedly high false positive rate. The median across all projects is 0%, and only four of the Jiras show a median between 0 and 1%. Inconsistent properties appear to be more of a problem in Maintenance issues, which makes sense given the increased number of stakeholders and stakeholder groups who are involved with Maintenance. With these increased roles, knowing when and how to update the fields would become less certain, and perhaps not even possible for some stakeholder groups (such as people external to the organisation). Analytically, it would seem that inconsistent properties are a problem when they occur, but empirically we can see that this occurs very rarely. I also assume that once my naive preliminary script is updated to be more intelligent, the number of detected violations will be even lower.

8.4. Summary

In this chapter, I implemented 18 different algorithms to assist in the automated detection of violations of Best Practices for ITEs. These algorithms are the starting place for ITS plugins that support practitioners in real-time and within their working context. With these algorithms, researchers can now challenge and improve the state-of-the-art for these Best Practices. Additionally, the results presented in this chapter can serve as a starting place for comparative studies looking to either improve on the algorithms, as well as inspiration for areas where case studies could investigate these phenomena.

With the algorithms constructed and applied to a real dataset, the next step is to consider the tooling they would be a part of. The logic has been written, but product design and integration in practitioner settings is a difficult task. How can these algorithms be integrated? When should this information be presented to practitioners? How much control should be given to users? I address these questions with tooling recommendations in the next chapter.

Part IV.

Outlook Recommendation

Chapter 9.

Tool Support for Best Practices in Issue Tracking Ecosystems

> We become what we behold. We shape our tools and then our tools shape us.
>
> Marshall McLuhan

In this chapter, I present a number of recommendations and guidelines for creating recommender systems for ITE Best Practices. In Chapters 7 and 8, I presented a catalogue of ITE Best Practices and algorithms to detect violations of those Best Practices, but those are all in isolation of each other and in isolation from their solution space. One of the findings from previous chapters is that ITE Best Practices (and the problems that they address) are context-dependent, which is captured in the ontology presented in Chapter 6. This leads to the logical conclusion that the solution space for recommender systems for these Best Practices is also context-dependent. Recommender systems for ITE Best Practices need to consider and allow for the context factors to guide their implementation and usage.

In this chapter, I focus on overall tool design for such a complete system that addresses the context-dependent nature of ITE Best Practices. I present descriptions of desired attributes of recommender systems, and in some cases I have already developed preliminary tools and demos to showcase some of this functionality. I then present these designs to industry participants to get their feedback on visual design and proposed usage of such tools. The primary feedback from the interviewed industry participants is confirmation of the usefulness of the context-dependent features. This includes the configuration screen, which allows individuals, teams, and organisations to set different preferences for their Best Practices, including the tuning of certain variables (not just binary "on" or "off" for each Best Practice). Another finding is that certain context factors play a larger role. For example, the role a stakeholder has within the organisation is an important context factor for not only which Best Practices they prefer, but also entire tool design choices. For example, I found that developers prefer integrated feedback directly within ITSs while creating or editing issues, whereas managers prefer summary tools that give reports across the ITS.

This chapter contributes both design aspects of recommender systems for ITE Best Practices, and empirical feedback on these design aspects. Researchers can take this work and directly build on it with their own designs and future studies on our designs. Practitioners can already implement these designs within their own systems to report, understand, and enforce Best Practices within their organisations.

Publications. This chapter contains portions of my collaborative publication, some copied verbatim [184].

9.1. Research Methodology

My primary objective with this chapter is to prescribe and validate recommendations for tooling that supports ITE Best Practices. To prescribe recommendations, I will use a deductive approach, applying the collective knowledge presented in this thesis. I'm not seeking to provide empirical evidence, but rather to act as a knowledge expert on the topics presented in this thesis, and therefore provide value in the form of final recommendations. To validate the recommendations, I sought to gather insights about the potential usefulness of the recommendations. For this, I interviewed 26 practitioners who use ITSs, with a focus on showcasing the recommendations, features, and screenshots from this chapter. I first asked about tooling in general to collect ideas without the bias of a specific tool or feature. I then presented four screenshots to guide the discussion and stimulate in-depth answers: (1) a configuration screen for the detection of individual smells, (2) a dashboard summarising the detected smells across the ITS, (3) a detailed view showing detected smells for the currently viewed issue, and (4) an issue-link graph visualisation highlighting link smells [171, 172]. I also asked if the shown features could address the problems and smells discussed earlier in the interview. I encouraged participants to share all thoughts on software tooling to support addressing ITE problems and smells, particularly when manifested as smells.

9.2. Recommendations for Tool Support

> **Nested Configurations.** Use nested layers of configuration files to accommodate the layers of context that affect the applicability of ITE Best Practices in industrial settings.

ITE Best Practices are context dependent (see Chapter 6), and some of those contexts are naturally layered. For example, organisations are composed of teams, and teams are composed of people. Each of those contexts may affect whether to apply a Best Practice or not, and each of them are likely to affect the tuning of the Best Practices. For example, an organisation may adopt the "Assignee Bug Resolution" Best Practice (the bug assignee should be the one to resolve the bug; see Table A.13), but they have a contractor team based in another country, and they prefer that someone internal always resolves the Bug Reports, regardless of who is assigned. In this case, the context of that particular team is in disagreement with the context

of the organisation for the application of this Best Practice. However, instead of viewing it as a disagreement, it would be better to imagine that the team context is a sub-layer within the organisation, and so just this one team needs an exception to the organisational decision. This is where one can apply the power of nested layers of configuration for ITE Best Practices.

Configurations outline whether a Best Practice applies (should be automatically checked for violations) as well as the tuning of the Best Practices where that is relevant (e.g., "minimum number of words" for the Best Practice "Sufficient Description"). A basic configuration file could be in a simple format such as JSON, with a "key: value" system. The nesting of contexts can be applied by having different configuration files that set the same parameters. Then, once an organisation has decided on what constitutes a context layer, and which layers are nested within each other, the configuration files can hierarchically override each other as the nested layers of context are applied. In our example above regarding the Assignee Bug Resolution, the organisation would have a organisational-wide configuration with the Assignee Bug Resolution Best Practice *enabled*, and the specific contractor team would have their own configuration where the Assignee Bug Resolution Best Practice is *disabled*. Since the *team context* is more specific than the *organisational context*, the team configuration overrides the organisational one. However, the team configuration file in this example would only set this single configuration, and the rest would be left blank, such that they would adopt whatever configurations apply from the organisation. The team context is not independent of the organisation, it is just more specific and therefore *can override* specific recommendations, *where needed*.

Conceptually, what constitutes a context layer, and which layers are nested within each other (or perhaps only partially overlapping), is a complex problem in and of itself. From a technology perspective, it is easy to implement these configuration files, and to write an override method that consumes all files and outputs the specific configuration for a given a context. What an organisation must decide, as a process decision, is which contexts are allowed to have their own configuration files, and what is the hierarchical layering of these contexts within their given organisation. As a starting point, I recommend the following context configuration files, listed in order of increasing precedence: **organisation, team, project, sprint, individual**. However, there are many good reasons to use additional configuration files, as well as different precedence orders. In many customer-centric organisations, the customer might have a say in organisational processes and the quality therein. For example, an organisation could tune the Timely Severe Issue Resolution Best Practice to four days, but a particular customer might need a timely severe issue resolution of just one day. In this case, the customer should have a configuration file that is likely just after "team" in the precedence.

Figure 9.1 shows a prototype of the Best Practice configuration screen, as developed by Nina Prediger [184]. For each Best Practice, the user (or team, organisation) can enable or disable it, tune it with particular best-practice-specific settings, and even give it a weight (importance). This configuration screen does not show the layering of configurations, but that would likely be a separate interface only designed to managed by one person within the organisation. The

configuration screen presented in Figure 9.1, however, would be utilised by every person in the organisation who wishes to adapt their own configurations, or the configurations of their project, team, or customer, on behalf of those entities.

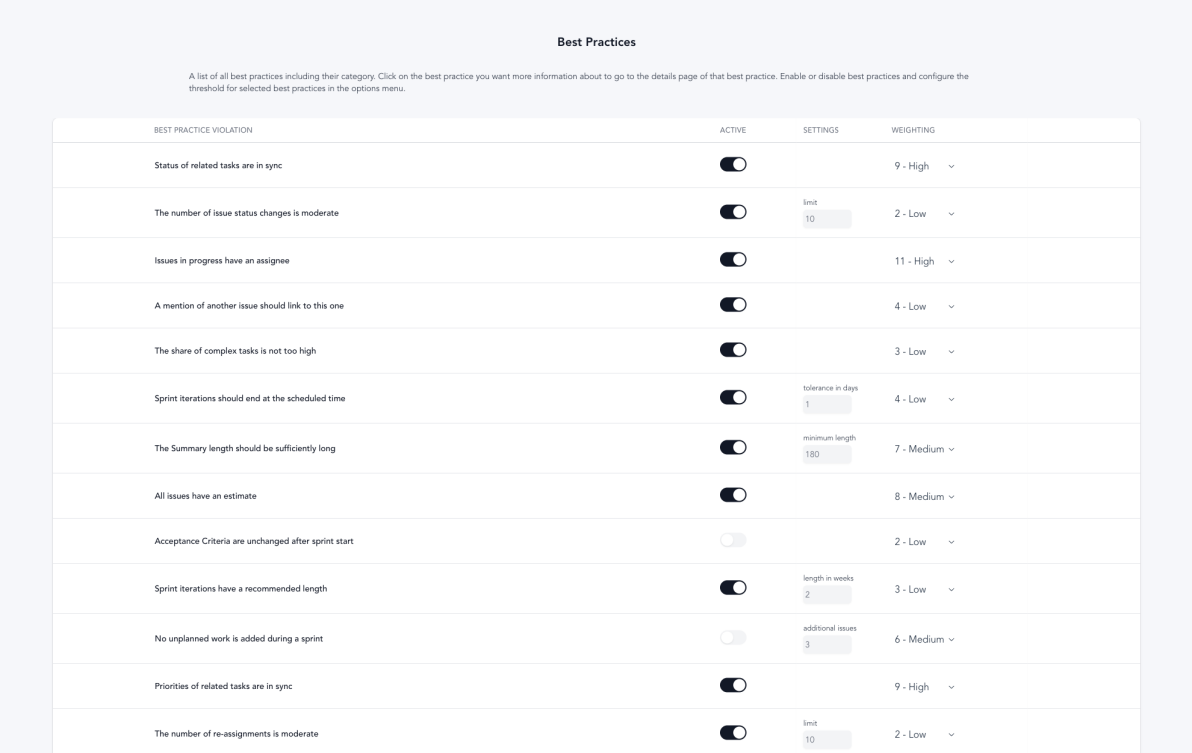


Figure 9.1.: ITE Best Practice Configuration

Cold Start using ITE Best Practice Historical Analysis. Use historical analysis of your ITS to understand which ITE Best Practices to address within your organisation.

The initial catalogue of ITE Best Practices that I presented in Chapter 7 already has 40 Best Practices, and the hope would be to grow this list to account for all quality-based methods for improving ITEs. With so many potential catalogue items, it may be confusing for an organisation to chose and start implementing these Best Practices. A data-driven approach to this decision would be for the organisation to first run the algorithms I presented in Chapter 8 on their ITS. These results would inform them which Best Practices are being most violated within their organisation, and therefore which to address first. It could also be the case that certain Best Practices have many reported violations, but the organisation could decide that these are in fact not violations within their context and defined processes. Either way, this would ground their understanding of ITE Best Practices in their specific context, and be the starting point for organisation-wide decisions.

Just-in-Time Feedback for ITS Users. Deliver feedback on ITE Best Practices immediately through ITS plugins that annotate issues.

Remembering all the Best Practices implemented in a *given context* is not feasible for the average ITE stakeholder, let alone *all contexts*. To train their memory and provide immediate feedback, I recommend utilising ITS plugins to prompt ITS stakeholders at the exact moment they have violated a Best Practice implemented by their organisation. Here are some plugin ideas that could provide immediate just-in-time feedback on ITE Best Practices:

Sufficient Description When creating or editing an issue, check the description length when the user tries to "save" the issue, and prompt them with a warning if the description is too short. At the extreme, it could also block them entirely from saving the issue until the description is sufficiently long.

Set Bug Report Assignee When creating or editing an issue, check if the assignee field is empty when the user attempts to "save" the issue.

Set Bug Report X Same as "Set Bug Report Assignee" above, but now applies to the other Best Practices of this structure.

Consistent Properties When editing the description or adding a new comment, check the text when the user tries to "save" their work, and prompt them with a warning if any issue field names are found in their text (which is a sign that they are adding information to the description or comment that should instead be changed directly in the properties).

Bug-to-Commit Linking When a user attempts to change the resolution to "fixed" on a Bug Report, check if the Bug Report has the commit linked in the description or custom field for commit links. If not, warn or stop them.

Those are just a few examples of just-in-time feedback that can be delivered to ITS stakeholders through native ITS plugins. With the use of NLP techniques, more just-in-time checks can be performed on the textual fields. Research applying NLP techniques has produced promising results in the last couple of decades [82, 165, 185, 206, 248]. Overall, there are many opportunities to support ITS stakeholders and guide them towards Best Practice conformance.

Issue Report for ITS Users. Deliver feedback on ITE Best Practices through an ITS plugin that annotates individual issues with a report of which Best Practices are violated.

There are many reasons Best Practices could be violated, including when an organisation adopts a new Best Practice, which may already be violated by thousands of existing issues. It is not reasonable to assume that stakeholders will immediately update all issues to conform to the new Best Practices. Instead, an ITS plugin can be used to create a small issue report in the properties section of each issue. Many of the Best Practices are applied to a single issue, which

means it is possible to give a report on the quality of that issue, regarding how many violations it has. When someone is working on that issue, be that creation or editing, this report would be immediately visible to them, in the context of their work.

Using this issue report, relevant stakeholders who have navigated to a particular issue will see the issue report, and they can make an in-the-moment decision regarding whether to improve the issue or not. This has the benefit that stakeholders will have the context of the issue in mind when they see the reported Best Practice violations. Figure 9.2 shows an example of what this could look like, in the bottom right of an issue, labelled here as "Issue Health".

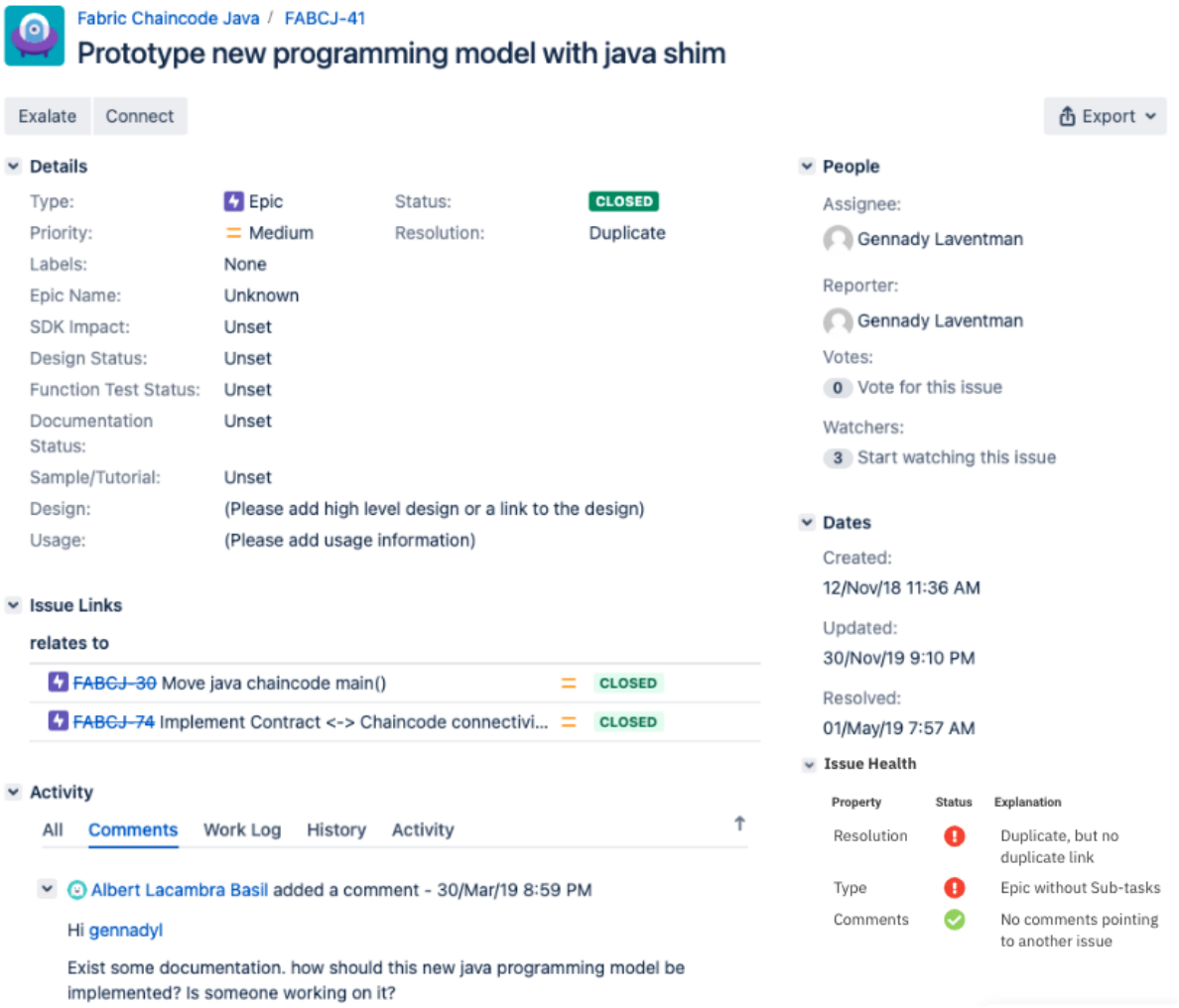


Figure 9.2.: ITE Best Practice Issue Report

> **Dashboard for Managers.** Summarise the conformance to ITE Best Practices across the ITS by creating a dashboard for managers.

While just-in-time feedback and issue reports for ITS users are appropriate to help guide

them, it is missing an ITS-wide perspective on the status and progress of Best Practice conformance. Over time, an organisation should see progress towards a higher-quality ITS through more conformance to the Best Practices they choose. For this, I recommend implementing a dashboard that summarises the conformance to Best Practices across the ITS. Figure 9.3 shows a prototype of the Best Practice dashboard, as developed by Nina Prediger [184].

There are three key features within this dashboard: summarisation, filtering, and historical records. It is important to summarise the conformance in a single value, such that a quick judgement can be made on the current progress. It is essential that the dashboard supports filtering to specific contexts (e.g., projects and sprints) so that managers can dive into the data, to contextualise and investigate the specifics behind the summarisation that is reported. Finally, keeping historical records of the summarisations presented allows managers to see progress over time, instead of just a single current summarised value. For the ITS Jira, where historical records of all issue changes are already stored, it is possible to see what the summarisation value would be at any point in the past. Combined, these three features would allow a manager to see the current status, set goals, and see progress over time.

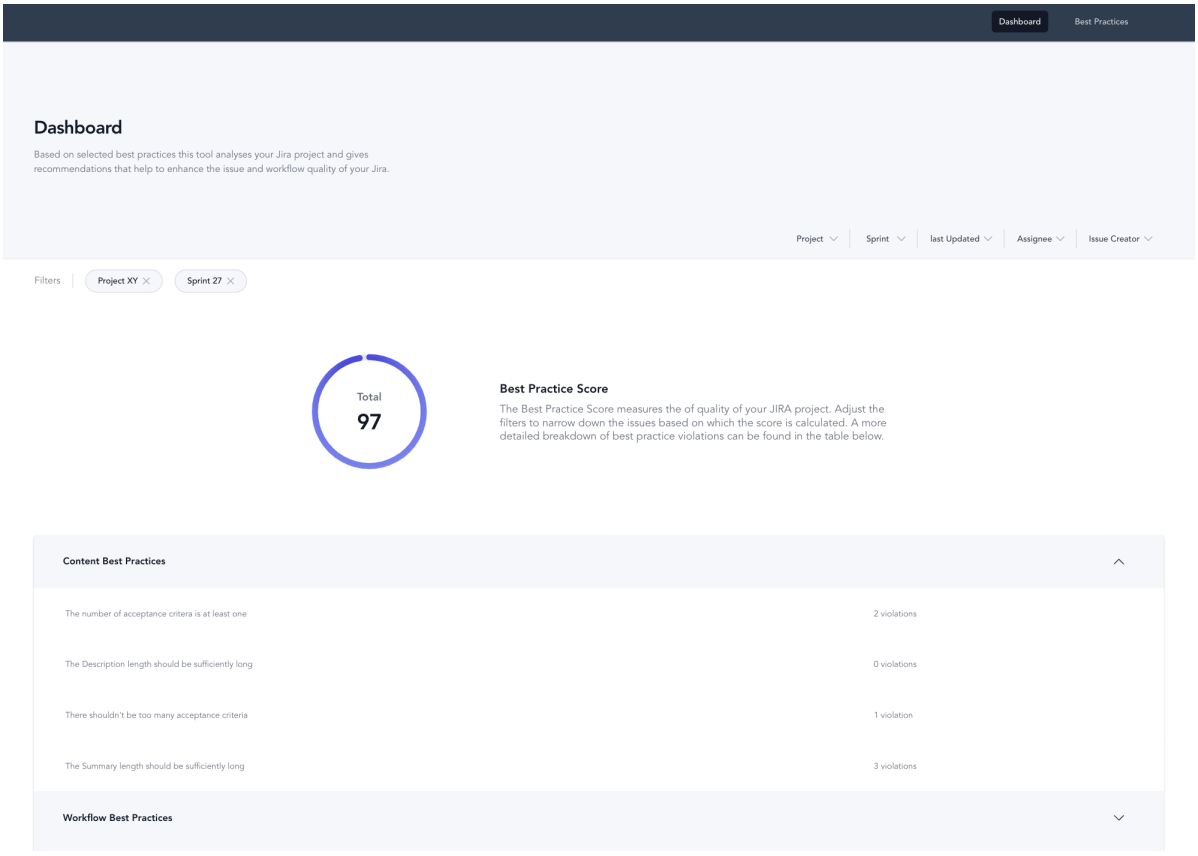


Figure 9.3.: ITE Best Practice Dashboard

9.3. Practitioner Feedback on Tool Support

I first asked participants about any *general tooling insights* they have, and then showed them the screenshots of our prototype tools to get feedback on specific ways of *enhancing ITSs*. The participants described that tool support must be easy to use, integrated well into their workflow, and provide reliable results. By meeting these requirements, they believe a tool could help improve the efficiency and effectiveness of the ITE process.

9.3.1. General Tooling Insights

Smell Detection Tooling to Support Existing Meetings. Participants mentioned that smell detection tooling could be used to support the quality of their ITS in existing meetings such as triage and sprint planning (P01, P08, P11, P13, P14, P20, P25). Multiple participants mentioned that triage meetings are essential to organising ITSs and ensuring that issue reports have the necessary information for developers to act on them. These meetings are already a way to ensure that issues in the sprint adhere to a quality standard and can be resolved. Participants mentioned that smell detection could support these meetings by highlighting potential problems. P14 also mentioned the potential to support retro meetings: "it would be nice to look at a dashboard and say we were 50% worse in this smell".

Prevention over Detection. Participants said that instead of detecting the smells, preventing them from happening might be more useful; otherwise there would still be an overhead of fixing smells after the fact (P03, P05, P08, P09, P13, P16, P17, P24, P26). Furthermore, one participant noted that a post-mortem of the issue reports might not be the correct way to fix them, but that the user should instead be alerted during the creation or editing. For example, P05 said that "smells should be prevented during issue creation, for example: if you want to close an issue as duplicate, you get a pop-up if there is no link". P08 agreed: "it would have to be directly integrated into [the ITS]; a separate tool would not make sense". The participants noted that an essential requirement for adequate tool support in ITEs is the direct integration into their ITS workflow. This means the tool should be easy to use and not require too much overhead while creating or modifying issues; otherwise, it will not be used. P25 said, "if you are not able to really address these smells, you have to re-think whether these kinds of tracking systems, properties, and linking are working for you. It reminds you to rethink different things".

Detecting Link Smells. 21 out of the 26 participants believe that link detection would be helpful, P18 and P20 were unsure about the usefulness and P04, P24, and P25 did not consider link detection helpful. P01, P02, P07, and P16 voiced that link detection would help locate issues or relevant issue reports in the ITS. A generated graph or tree representation was also suggested by some participants to achieve a high-level view of the connections between projects or gain an overview or understanding of the context. P24 said, "suggested links need to be reviewed and when it turns out that they are not actually related, [extra work was created]". They would "rather miss a link than have to inspect unnecessary ones".

Tooling Pitfalls to Avoid. Participants specifically mentioned that smell detection would be more useful for managers and not developers (P02, P05, P07, and P09). They also mentioned that the tool should not generate too many false positives because the additional noise is "annoying" and users do not want to "sift through a large collection of smells". They expressed concern that the user might receive too many notifications (P06, P07, P12, P14, P15, P21, P22, P24, P26). P07 stated the system "should not spam too much; if the information is not too obvious then it would be very helpful". Notifications can be annoying, especially outside the ITS, e.g., e-mails [154]. P14 also cautioned that there might be social impacts of such a tool, making users feel bad instead of helping them improve. Additionally, participants mentioned that the tool should not obstruct the workflow, as it creates more overhead. P13 said that "enforced processes are not good" and P22 said "it should not completely block [your workflow]". Several participants voiced their scepticism toward automated processes (P07, P08, P12, P15, P18, P21, P23, P24). They said that the results need to be transparent, interpretable, and manually reviewed. However, P24 reported from experience that "suggested links need to be reviewed, then it turns out that they are not actually related", which led to the creation of extra work. They would "rather miss a link than have to inspect unnecessary ones". There also must be a general acceptance of the tool in the team for it to be used. Three participants (P05, P15, P23) were unsure about the usefulness of smells and participant P09 stated that it would not be useful for them. It was noted that some kinds of smell detection would likely be more useful for managers than developers. Additionally, one participant noted that smell detection can be counter-productive for developers.

9.3.2. Enhancing Issue Trackers

(1) Configuration. We already saw that most smells are not perceived universally as problematic, and some are indeed part of the good practice. I showed the participants a screenshot of a smell configuration page, containing a toggle for each smell (and some had a configurable value). Almost all participants said that a configuration is a must-have, a tool "must be customisable because some smells would not work or would constantly [be detected]" (P03). P10 mentioned that the tool could be misused and hinder more work than help if it is not configured properly. P13 cautioned that "deactivating smells is a nice [feature], but [they] need to understand and document why smells are deactivated". P26 also added that a *default* configuration is needed. Finally, P05 mentioned that they would like "syntax to write their own smells".

(2) Dashboard. I showed the participants a dashboard presenting the overall "health" of the ITS. Multiple participants commented that a dashboard has more value for managers, with little value for developers. It was also noted that it should not be the absolute values but the trends over time. P14 said, "I don't care about the actual amount, I care about the trend". P21 said, "it would be nice to see the change over time with the health, so it is not only the static view that is important, but the development over time". P24 said that tools like this

should be used with a goal in mind because "'optimising for green isn't always good" and that the metrics shouldn't become the targets as they can "distract from what the customer wants". P14 cautioned that "metrics can get in the way of non-tracked benefits people bring; reducing people to metrics might distract from the important social dimensions of the workplace". "As a manager, I feel this could be a great way to connect with the developers" (P20).

(3) Detailed Issue View. I showed the participants an issue view with integrated feedback on detected smells for this individual issue. Developers seemed to prefer smell detection for individual issues rather than in an ITS dashboard. P17 said that "[the information] needs to be pre-digested into actions being requested of the developers; they can then decide to do it, or not" and P02 said "but they don't need the stats, just the lists of smells and where/how to fix it". P06 said that "issue page hints could be very helpful".

(4) Link Graph Visualisation. I showed the participants a graph view of interlinked issues, including which link smells each issue has. Smell lists and visualisations can help detect violations or identify relevant links, but finding the right balance between the two is essential. Participants mentioned that lists are sufficient and visualisation is not needed for managing links. Visualisations may be preferred at higher levels (e.g., epics) but may be overkill at lower levels (e.g., subtasks). A high-level abstraction from lower-level links may be sufficient, such as by extracting "related" epics based on the presence of related user stories, tasks, or bugs. P22 said that this also depends on the usage of links: "depends how you work with it, what kind of project you are working in, and if a team uses a lot of issue links: then this would be helpful".

9.4. Discussion

Users are interested in software frameworks that can detect violations of Best Practices. These tools may be particularly useful for managers, but they must be integrated into users' workflows. While visualisations are nice, it may not be necessary for the detectors to be valuable. The users interviewed had different and sometimes conflicting priorities and needs regarding the tools' functionalities. Existing links and larger issue graphs are more important for the R&D team lead and product managers, who may benefit from visualisation, confirming the results of Li and Maalej [133]. Tooling could unearth underlying process problems, for instance through violation detection. Often missing or changing properties or too many options might lead users to rethink their processes and finally lessen the bloat by revamping their workflow.

For efficient ITS usage, finding the right balance between restriction and freedom is essential. Restricting users too much can reduce efficiency if they become frustrated with the process, but if the degree of freedom is too high and things are not checked, many mistakes can occur that lead to problems in the future. Restriction and freedom may depend on the specific needs and processes of the team or organisation using the ITS. Herraiz et al. [101], for example, argued for simplifying the issue report form in Eclipse. Too many options to choose from make it hard for reporters to choose the correct one and should, thus, be limited.

In large ITSs, automation is necessary due to the overhead involved in tasks such as reminding someone to update properties (e.g., status, due date, estimation) or updating properties based on the status of linked issues (e.g., if all sub-tasks are closed, the epic should be closed). While Jira automations exist [112], they seem not to be widely used. One reason could be the missing awareness of practitioners of these features. Another reason could be the overhead of configuring and updating these rules. Supporting or recommending specific automation rules based on ITE usage patterns need to be investigated. Overall, there appears to be a need for a (smart) guide on how to use ITSs efficiently.

9.5. Summary

In this chapter, I recommended five core features for tooling support for Best Practices in ITEs. I then interviewed practitioners to get their feedback on these features. Practitioners agreed with the benefits these features bring to ITEs. Depending on the context of the practitioner, whether manager, developer, or large company, they had slightly different perspectives on the usefulness of these features. Overall, these findings showcase the potential for improvement in ITEs through specific tooling features. In this chapter, I started the discussion regarding recommendations for future work for this research, with a particular focus on practitioner impact through tooling support. In the next chapter, I summarise the contributions of this thesis, and outline future work areas specifically aimed at researchers.

Chapter 10.

Conclusion

> Nach dem Spiel ist vor dem Spiel.
>
> German Expression

In this thesis, I addressed three primary gaps through four parts, across ten chapters, producing nine central contributions (see Fig. 1.1). I investigated the difficulties practitioners experience with ITSs by interviewing 26 practitioners. I collected and analysed 16 Jira repositories to better understand the complexities that exist within ITSs. I empirically formed the Best Practices Ontology for ITE, created an initial catalogue of 40 Best Practices, and implemented 18 algorithms to automatically detect Best Practice violations. Through these empirically driven investigations, I have made progress towards addressing the three primary gaps. We now know considerably more about the difficulties encountered by practitioners using ITSs, as well as the complexities within ITSs. We also now have context-dependent recommendations, and a theoretical structure to further improve this area of research and practice.

10.1. Threats to Validity

In this thesis, I conducted various data gathering and analysis techniques. In this section, I will summarise the threats to validity across the techniques in aggregate. Where necessary, I also discuss specific concerns with certain techniques applied in particular chapters. This thesis collects and processes both quantitative and qualitative data, with a particular focus on understanding and summarising large amounts of textual data.

The collection of the Jira dataset introduced a potential sampling bias into the results, as I did not get a "random sample" of all Jira repos to conduct this research on. However, I feel this bias is outweighed by the benefit this large dataset provides, as I do not know of any larger set to which I could apply stratified representative sampling. In the end, the dataset is across 16 Jiras (including world-leading software organisations such as Apache, RedHat, and Spring), with 2.7 million issues and 30 million evolutions. Given the large dataset, I believe the claims made in this thesis indeed represent a larger understanding of public ITSs in practice. Additionally, other ITSs may have other features and be used differently, but as noted in Chapter 4: "Jira is

by far the most popular tool in the ITE and agile project management markets".

Observer bias represents one main risk for interview studies, where the interviewers elicit the statements they are expecting or hoping for. I took multiple measures to mitigate this risk, as discussed in Chapter 3. Notably, I explicitly encouraged participants to disagree and share their personal opinions. There were also two interviewers in each session, we posed the questions as neutral as possible, and we exposed participants to information only when they had to see it. Due to participants' disagreements as well as the heterogeneous perceptions gathered, this bias appears minor. I might have misunderstood the interviewees or missed important points in my reporting, particularly as I did not record the sessions (for the criticality of the discussion and for better spontaneous engagement). To mitigate this risk, both interviewers took notes and asked for clarifications whenever needed. We also discussed each interview session afterwards to mitigate the threat of memory recall. Finally, our analysis of the notes was conducted by three authors (for each interview), is fully documented, and each sentence traced to the findings.

The interview sample includes only 26 experienced software practitioners. I think that this is a rather large sample, comparable to seminal qualitative studies in SE. The sample provided enough diversity and redundancy to derive the findings. However, I refrain from claiming generalisability of the results to software practitioners, and this never was the goal. Nevertheless, the diversity of the sample, involved companies, and redundancy of observation give me reason to believe that the overall trends hold true. To achieve quantifiable and generalisable results, follow-up studies (such as surveys or experiments with practitioners) would be needed. The qualitative findings serve as starting hypothesis and variables to measure for such studies. Similarly, the results are based on the subjective statements of practitioners. In theory, what people do might diverge from what they say. Therefore, triangulation with observation studies or artefact analysis will likely lead to stronger evidence and more insights.

This thesis relies heavily on the use of qualitative methods to interpret and categorise real-world phenomena (e.g., issue type themes in Chapter 4 and the information themes in Chapter 5), which can result in threats to construct validity. To mitigate these threats, I followed strict qualitative guidelines designed by prominent methods researchers and analysed thousands of data points to achieve strong resonance and saturation. Where possible, I released the analysed datasets, including intermediate steps. Additionally, I referenced outside materials where available, triangulating my interpretations with explicit descriptions of these constructs. By following these methodological guidelines, I am confident in the reliability of the findings.

10.2. Summary of Contributions

Cross-Study Finding that Context is Key. My findings across the interview studies and historical analysis highlight that "context is key", confirming that context plays a central role in shaping ITEs, including problems and solutions. It is no surprise to SE researchers that context plays an important role in a SE concept. However, ITEs have largely been studied and reported

without the context playing a central role in any of the experimental design, the reported factors for case studies, or the statements made about findings. This means that studies of ITEs are limited to the exact (often unreported) context in which they are studied, and findings are not contextualised for practitioners to know when and how to apply them in their own context. The interview studies revealed that problems experienced by practitioners are context-sensitive (occurring under some conditions and not under others), and the impact of these problems was also context-dependent. Some ITE problems and solutions were also in conflict with each other, and their prioritisation was dependent on the relevant context factors. The results of the historical analyses in Chapters 4 and 5 show a diverse usage of Jira, with different SE activities and information types emphasised in different contexts. To treat ITEs as universal would be a mistake, and for specific empirical investigations it would introduce major flaws to construct and conclusion validity. My findings highlight the importance of understanding, declaring, and investigating context factors when researching ITEs.

"Issue Tracking Ecosystem (ITE)" as a Fundamental Concept. I introduced the concept of an Issue Tracking Ecosystem (ITE): "an ITS tool, and all surrounding contextual factors that affect the ITS, including the stakeholders who interact with it, the company they work for, the team they work with, and the project they are working on." There currently exists many terms to describe tools such as "Jira", and they are both manifold and too narrow. Existing terms focus on the tool itself, rather than the larger context that surrounds, affects, and is affected by the tool. The concept of an ITE includes the environment around the tool, thus encapsulating and unifying what the term means. The term unifies and expands on the existing term usage regarding the rhetoric around ITSs, issue tracking systems, and bug trackers. The benefits of "ITE" as a concept goes beyond just addressing these limitations, and include drawing attention to the surrounding ecosystem, and encouraging an explanation when a researcher does not account for these ecosystem factors in their research.

An Understanding of Practitioner Problems within ITEs. I interviewed 26 practitioners who regularly interact with ITEs and produced a list of commonly reported problems. ITSs are commonly reported in pop-culture as difficult to use, but the studied details of such difficulties are lacking. We have lots of research regarding specific problems within ITEs, such as Bug Report quality [27, 250] and the correctness of "requirements" issue types [233, 240]. However, these pinpoint analyses are often not focused on the problem itself (e.g., its existence and the impact on industry), but rather trying to solve the problem. I found a wide variety of problems across three major categories: ITE information, ITE workflows, and ITE organisational aspects. Of particular interest was the trade-offs that occurred between potential solutions to one problem, that would then increase the likelihood or impact of a different problem. For example, one of the most common problems reported is "workflow bloat", in which the ITS workflow involves so many mandatory steps, that practitioners felt either overwhelmed or frustrated in the amount of time they had to dedicate to just interfacing with the ITS. However, an equally prevalent problem was people not using the ITS properly, and many practitioners

recommended that the ITS should enforce *more* on the user to solve improper ITS usage—which would then also increase the amount of workflow bloat. Overall, the findings highlight that even today, after decades of research into ITEs, there are still many problems to address.

Dataset of Issue Tracking Systems. I collected and published data from 16 public Jira repositories containing ~2000 projects, 2.7 million issues, and 30 million evolutions [164]. While GitHub is a popular, public, and well-studied resource for ITSs, it is also a code-first platform, and lacks certain customisation features available in Jira. According to 6Sense [1], Datanyze [46], and Enlyft [55], Jira is by far the most popular tool in the ITS and agile project management markets. However, finding Jira Repos to analyse is difficult because Jira is a private tool hosted by individual organisations. This Jira dataset that I published is the single largest public collection of Jira repositories that exists. Other datasets do exist, but they are either subsets of my dataset[1] or considerably small (1–10k issues) custom sets that have some other special data attribute (such as custom labelling). This dataset has been viewed 10,000 times and downloaded over 2,700 times,[2] showing the interest that exists for such data.

Data-Driven Characterisation of ITSs. I investigated the Jira dataset and revealed key information about practitioner usage of Jira, including the activities that are conducted within ITSs, the information that is managed, and the evolution that occurs. ITSs have been studied for many decades, but largely from specifics angles, without a holistic perspective. While there are many specific problems that that should and have been studied, a holistic perspective provides a comparative summary that allows for a complete picture of the system under study. My investigation revealed three high-level activities in ITSs: Requirements, Development, and Maintenance. For each, a subset of artefacts can be found, including Epics and User Stories for Requirements, Tasks for Development, and Bug Reports for Maintenance. Requirements issues account for ~30% of all issues (median % issues per project), Development accounts for ~20%, and Maintenance accounts for ~50%. My investigation also revealed different information types within ITSs: Content, MetaContent, RepoStructure, Workflow, and Community. My analysis then revealed how often these information types evolve, including the average of ~8 evolutions per issue, and most evolutions occurring to Workflow (~40%) and Community (~30%). Overall, my analyses reveals and details many intricate inner workings of ITSs that confirm some existing assumptions about ITSs, and challenge others.

Best Practice Ontology for ITEs. I created an ontological structure for ITE Best Practices that collects and re-frames decades of research into quality aspects for ITE. The ontology provides a unified structure to research, discuss, and communicate quality attributes for ITSs. The ontology, while not proposed as theory, has many important constructs to be framed as theory by future work. I also contributed five propositions which should be scrutinised by future work looking to contribute theory in this area. The ontology has five sections and 16

[1] My dataset has the full Apache Jira Repository, while it is quite common to find Jira datasets that have a subset of projects from that Jira.

[2] `https://doi.org/10.5281/zenodo.5882881`

dimensions, with the sections being: Meta, Summary, Recommendation, Context, and Violation. The ontology acts as a guide for those creating ITE Best Practices, but also for those transforming or challenging existing ITE Best Practices.

Catalogue of Best Practices for ITEs. I formed 40 Best Practices from existing research into quality factors for ITEs. The catalogue acts as a starting place for both researchers and practitioners to build on. Future research can refute and add to the knowledge that they contain, in a structured way. Practitioners can review and apply the Best Practices directly in their ITEs. The catalogue was constructed using techniques similar to a secondary study. Primary articles were collected, and then key information was extracted and summarised in the ontological structure, producing the catalogue. The catalogue is incomplete, and requires future work to fill in the missing unknowns.

Algorithms Detecting Violations of Best Practices for ITEs. I collected and created 18 algorithms to automate the detection of violations to ITE Best Practices. These algorithms allow the immediate application of these techniques in industrial settings. They are also a starting place for cross-analysis research, applying the findings I presented with my dataset. The algorithms are all built for Jira data, but their application only requires data that can be found in most ITSs. My findings from applying them to my Jira dataset revealed a mix of well-followed Best Practices (low violation rate), and ignored or unimportant Best Practices.

Nested Configurations to Satisfy Contextual Needs. I introduced the concept of Nested Configurations for tooling solutions that address ITE Best Practices as a direct recommendation to deal with the importance and complexity of context factors in ITEs. ITE research needs to acknowledge, specify, and investigate ITE phenomena with context in mind, and once those context factors have been identified, practitioners need a way to apply them into their ITS tooling. Practitioners also require a way to adapt the tooling to their context-specific needs. The solution to these situations is nested configurations, such that layers of settings can adapt and conform to contexts—whether identified in research or practice. In Chapter 9, I offered a recommended structure for nested configurations, listed in order of increasing precedence: organisation, team, project, sprint, and individual. This means that an organisation should define their preference for Best Practices in their ITE, but a team should be able to override particular Best Practices if their context demands it. There are also many good reasons to use additional configuration files, as well as different precedence orders. Overall, the nested configuration concept can adapt to many—and perhaps all—organisation contexts.

10.3. Future Work

Future work in the area of quality factors for ITSs and ITEs can take many potential directions. Given the momentum created by this thesis, I am biased towards the further investigation, improvement, and dissemination of ITE Best Practices. However, there are other potential directions as well, including an entirely different conceptualisation of ITE or the creation of

a new (or transformed) structure similar to ITE Best Practices. For the sake of a focused perspective, I will describe the future work I see for ITE Best Practices.

Investigate and Characterise Specific Patterns of ITS Usage

We need research investigating and characterising specific patterns of ITS usage. My investigation of ITSs in Chapters 4 and 5 surfaced many findings, but the landscape of holistic information about ITSs and ITEs is still very limited. Specific SE process models tend to view ITSs as supporting Agile SE (see Section 2.1.2) or a Change-based form of RE (see Section 2.1.3), however, my intuition after working with ITSs for more than a decade [159, 161, 166] is that ITSs support notably specific and alternative forms of SE across all phases of SE. Much of the continued research in this area treats these systems as homogeneous, making broad statements about the findings and recommending simplified solutions for practitioners. Additionally, much of the tangential research views ITSs as merely simple tools to house more complex processes. All these assumptions, as shown by my results, are simply not true.

Regarding specific directions for future work, I recommend exploratory empirical research that is grounded in both historical ITS data and interviews or surveys for follow-up member checking. These exploratory investigations need to apply rigorous content analysis methods such as Thematic Analysis to form a coherent and consistent perspective on the data. I recommend applying an *inductive* Thematic Analysis with *rich overview* as the level of analysis, working to surface *semantic* themes. My assumption is that specific patterns of usage will emerge, which we can then member-check with industry participants, and eventually begin to generalise to other industrial contexts that apply similar processes. My expectation is that a clear set of patterns will emerge, which can be found across types of ITSs, in different ITEs. The work of van Can and Dalpiaz [233] is already conducting research in this direction: starting bottom-up, and investigating what exists in ITSs (inductively), instead of presupposing models of usage. I believe that similar continued efforts is the best way to fully understand and therefore offer the best recommendations for these industrial ITEs.

Investigate and Characterise Context Factors for ITE Best Practices

We need research investigating and characterising context factors for ITE Best Practices, to make these recommendations more specific, applicable, and adaptable. Related work in the area of context for tool support has highlighted that context is often dynamic and learned [95, 141]. My investigation into practitioner challenges with ITSs in Chapter 3 and structuring of existing research into ITE Best Practices in Chapter 7 show the importance and lacking of known context factors in ITEs. In addition to the knowledge we need on "specific patterns of ITS usage", we also need to consider the context factors that affect those patterns of usage. ITE Best Practice recommendations that work consistently for a given ITS usage pattern may not apply if specific context factors change. As discussed at length in Section 6.2.4, I believe

it is possible to remove all notions of subjectiveness when it comes to the application of ITE Best Practices because the "it depends" exceptions are just the result of hidden factors at play. While "smells" still have a valuable role in the SE landscape (both in research and practice), I strongly believe that a mature understanding of quality factors for ITEs involves elevating ourselves from smells to Best Practices—which account for context factors.

Regarding specific directions for future work, I recommend multiple-case study research with an embedded design (see Runeson et al. [195] Section 3.2.3). The cases should be different project settings within a largely shared context, and the units of analysis should be the different ITE Best Practices. The largely shared contexts grants similarities that simplify the investigation (e.g., same company and maybe even same team), while the different project settings (e.g., different project, sprint, or customer) grants experimental-design-like differences to focus the analysis on. The number of ITE Best Practices under study should be kept low to focus the analysis (max 3–5). The data collection methods should include data analysis of the live repositories, but more importantly, interviews, observations, and surveys of the people involved to understand the success (or not) of the Best Practices on their ITE. After considerable data collection, the data analysis should involve in-depth cross-case analyses to compare which shared context factors had similar impacts on the different Best Practices, and which separate context factors likely played a role in their respective successes or failures. Within the scope of a single embedded multiple-case study, it is reasonable to assume that meaningful results can emerge that outline the relevance and impact of multiple context factors for multiple ITE Best Practices. I believe that this form of targetted research would reveal many meaningful insights that further our understanding of both ITEs and the Best Practices that apply to them.

Validate the Initial Catalogue of ITE Best Practices

We need research validating the initial catalogue of ITE Best Practices, to increase the confidence in the extracted information and discover flaws in the ontological design. My creation of the ITE Best Practice Ontology in Chapter 6, followed by my creation of the initial catalogue in Chapter 7, produced an empirically grounded starting place for this conceptual structure. However, there is still much work to be done before any reasonable trust should be put in the results. The ITE Best Practice Ontology is designed to structure, support, and encourage well-formed theory in the area of quality factors for ITE. The catalogue is designed to offer a starting place for what these Best Practices could look like, and offer some amount of robustness testing for the ontology itself. However, the nature of these initial conceptual frameworks, including the design of my work, is such that follow-up research is needed to validate the results. In my case, there are two possible avenues for validation: either the ontology itself from a purely theoretical perspective, or any of the ITE Best Practices from either theoretical and practical perspectives. I will only discuss the latter avenue: validation of specific ITE Best Practices.

Regarding specific directions for future work, I recommend a mix of two techniques. First,

holistic single-case studies for practitioner feedback and investigation of loose casual effects of the Best Practice on meaningful outcomes. Second, historical data analysis for systematic investigation of Best Practice application, implementation, and intervention. By the nature of the initial formation of the Best Practices, it is not known which parts of each Best Practice need to be validated the most. For this reason, I recommend a more open and exploratory case-study-like investigation of the application of these Best Practices in industrial settings. I believe the quickest and most direct way to challenge (and therefore learn about) the Best Practices is to explore their full application. These lessons learned can then be directly applied to the Best Practices to confirm or update the Best Practices. I also recommend historical data analysis due to the rich nature of ITS data, including historical analysis of pre- and post-intervention outcomes. Many of the desired benefits of ITE Best Practices can be directly measured within ITSs, and therefore it is possible to algorithmically and systematically study the impact of Best Practices quantitatively. Quantitative analysis of sociotechnical outcomes has been repeatedly challenged in SE research [190], including the creation of productivity frameworks that consider a wide range of context factors and desired sociotechnical outcomes [68]. This should be considered when investigating Best Practices quantitatively, since what is "desirable" and what is quantitatively "measurable" may not intersect for a given ITE Best Practice. I believe a combination of these techniques will lead to more robust, validated ITE Best Practices.

Engineer a Complete Tooling Solution to Integrate ITE Best Practices into ITSs

We need to engineer a complete tooling solution to integrate ITE Best Practices into ITSs to streamline and support practitioner adoption of the Best Practices. My work has provided much of the foundational pieces required for a complete tooling solution. The Best Practices are provided by Chapter 7, the algorithms are provided in Chapter 8, and the tooling solutions are provided in Chapter 9. However, there is still much engineering work to be done. Product development requires additional work, including analysis of which product features need to exist, which user groups should be targeted, how to integrate into the various ITSs, which ITS to begin with, and which ITE Best Practices to start with. Of notable importance, is to consider how the tooling can integrate, communicate, and support the adoption and maintenance of the Best Practices. The integration of the violation detection algorithms is the most straightforward aspect, and the visualisation of the results of these algorithms is similarly approachable. Other aspects, however, can be much harder to integrate, and can easily add to the existing complexity and feature bloat that already plagues ITSs. For example, it is not clear how to support process suggestions made in the "Recommendation" section of ITE Best Practices.

Regarding specific directions for future work, I recommend to first engineer the algorithmic detection and display of violations, then build the system of nested configurations, then investigate classes of process recommendations and implement them one at a time—with extreme sensitivity to visual and process clutter. The algorithmic detection work has already begun,

with my algorithms presented and published in Chapter 8. All that is missing is the integration of these into a live system with real-time access to the ITS database, which requires custom connectors for each ITS data model. Once real-time algorithmic detection is built, then a custom display of these findings need to be built into the individual display of issues. To support the management perspective (as discussed in Chapter 9), a dashboard should be built to visualise the results across larger sets of issues. The implementation of a nested configuration system should be fairly straightforward, given the design description in Chapter 9. At any given point, the system needs to be aware of both the nested configurations, and the current context within the system. The result is a decision whether to apply the algorithms or not. Finally, the process recommendations are indeed unique, and require intense investigation. Examples of process recommendations include real-time suggestions while writing or editing issues, pop-ups that suggest things to users while using other parts of the ITS, required reading of agreements before performing certain actions within the ITS, and role-based restrictions on actions. To begin this process, my recommendation is to focus on in-the-flow and in-the-moment suggestions for ITS users. This brings information to users in the exact context it is intended for. Users can then make in-the-moment decisions whether they want to act. Other process recommendations can be considered and integrated over time. I believe that these combinations of tooling integrations will create meaningful and useful improvements to ITEs.

Investigate the Full Application of ITE Best Practices in Industrial Settings

We need to investigate the full application of ITE Best Practices in industrial settings to understand the true impact of applying such recommendations in industry. Considering the full culmination of my thesis work, the ultimate long-term goal is to improve the quality of ITEs overall. This goal requires the application of all future work items mentioned above, and then the application of the finalised and complete system in industrial contexts. For meaningful improvements to real use cases, we need a broad characterisation of specific ITS usage patterns. For repeated success in new environments, we need a broad characterisation of context factors for ITE Best Practices. To obtain confidence in the desired impact of individual ITE Best Practices, we need evidence of the success of the Best Practices from multiple case studies. For easy and repeatable integration of ITE Best Practices, we need tooling that supports and adapts to many ITSs and ITEs. Regarding specific directions for future work, I wish all the best skill and luck to the researcher that undertakes such a large, complex, and long-term mission. I believe that such a mission is possible, and would bring many needed improvements to the ever-growing world of ITEs.

Part V.

Appendices

Appendix A.

Full Catalogue of ITE Best Practices

In this appendix, I show all 40 Best Practices for ITE. First, I show the table of contents again, to act as a guide. Then, I list the Best Practices one at a time.

Table A.1.: Best Practices Catalogue - Table of Contents.

ID	Name	Objective	Source	Page
		— Issue Properties —		
BP01	Good Bug Report	Achieve good Bug Report quality by having the necessary elements.	[27, 250]	179
BP02	Atomic Feature Requests	Each Feature Request has only one request.	[98]	180
BP03	Assign Bugs to Individuals	Achieve good Bug Report quality by having the correct fields set.	[187, 188]	181
BP04	Sufficient Description	Provide a sufficiently long description.	[184]	182
BP05	Succinct Description	Issue descriptions should be succinct, particularly for requirements and development issues.	[137]	183
BP06	Avoid Status Ping Pong	Streamline issue state changes through the identification and removal of state cycles.	[10, 94, 184]	184
BP07	Avoid Assignee Ping Pong	Streamline issue assignee changes through the identification and removal of assignee cycles.	[10, 94, 184]	185
BP08	Set Bug Report Assignee	Achieve good Bug Report quality by always setting an assignee.	[184, 187, 188]	186
BP09	Set Bug Report Priority	Achieve good Bug Report quality by always setting a priority.	[187, 188]	187
BP10	Set Bug Report Severity	Achieve good Bug Report quality by always setting a severity.	[187, 188]	188
BP11	Set Bug Report Environment	Achieve good Bug Report quality by always setting the environment variables.	[187, 188]	189
BP12	Assignee Bug Resolution	The bug assignee should be the one to resolve the bug.	[187, 188]	190
BP13	Avoid Zombie Bugs	Foster a meaningful backlog by keeping issues alive or resolving them.	[94]	191
BP14	Active Bug Reports	Cultivate a meaningful open Bug Report backlog through regular activity or archiving.	[10, 187, 188]	192

(continued on next page)

Table A.1, continued

ID	Name	Objective	Source	Page
BP15	Bug Report Discussion	Achieve good Bug Report quality by encouraging discussion.	[187, 188, 218]	193
BP16	Respectful Communication	Maintain respectful discourse within issues and their comments.	[137]	194
BP17	Consistent Properties	Issue properties should always represent the most up-to-date information.	[137]	195
BP18	Good First Assignee	Achieve good Bug Report quality by assigning the best developer first.	[187, 188]	196
BP19	Stable Closed State	Increase the confidence that once a bug is closed, it will stay closed	[187, 188]	197
BP20	Timely Severe Issue Resolution	Address severe issues within a defined timeframe.	[94, 184]	198
BP21	Issue Creation Guidelines	Guide users how to create good issues.	[98]	199
BP22	On-Topic Discussions	Maintain topic-related discussions on Feature Requests.	[98]	200
		— Issue Linking —		
BP23	Bug-to-Commit Linking	Achieve good Bug Report quality by fostering traceability between bugs and resolving commits.	[187, 188]	201
BP24	Link Duplicates	Maintain good Bug Report quality by referencing related closed bugs.	[187, 188]	202
BP25	Minimal Link Types	Reduce redundant link types to simplify linking.	[137]	203
BP26	Record Links	Record all issue links using the linking feature, not in comments or in the description.	[137]	204
BP27	Realistic Dependencies	Catch and resolve unrealistic dependencies in link networks such as cycles.	[137]	205
BP28	Singular Relationships	Simplify ITS dependencies by limiting 1-to-1 relationships to a single link.	[137]	206
BP29	Connected Hierarchies	Connect hierachies with links to view the full picture.	[137]	207
BP30	High-Dependency Bugs First	Foster a low-dependency ITS.	[94]	208
BP31	Search Reminders	Decrease the number of duplicate Feature Requests by prompting users to search for existing ones first.	[98]	209
		— Issue Processes —		
BP32	Ordered Product Backlog	Maintain an ordered product backlog for meaningful product direction.	[53, 54]	210
BP33	Team-Produced Work Estimates	Create accountability between teams and their tasks by allowing them to make their own work estimates.	[53, 54]	211
BP34	Story Points Over Hours	Manage expectations and work towards correct predictions of work effort required per sprint.	[53, 54]	212
BP35	Estimate all Items	Estimate all work items before starting a sprint.	[221, 222]	213

(continued on next page)

Table A.1, continued

ID	Name	Objective	Source	Page
BP36	Avoid Unplanned Work	Focus on planned work during a sprint, to achieve the agreed commitment.	[221, 222]	214
BP37	Recommended Sprint Length	Maintain a core element of Scrum by using 2–4 week sprints.	[53, 54]	215
BP38	Consistent Sprint Length	Maintain constant and simplified velocity by using the same sprint length for every sprint.	[53, 54, 221, 222]	216
BP39	Use Acceptance Criteria	Improve user stories by using acceptance criteria as a measure of completion.	[184]	217
BP40	Limit Acceptance Criteria	Improve user stories by limiting the number of acceptance criteria.	[184]	218

Table A.2.: BP01: Good Bug Report.

Summary

Objective: Achieve good Bug Report quality by having the necessary elements.

Motivation: "Bug reports are vital for any software development. They allow users to inform developers of the problems encountered while using a software. [...] However, Bug Reports vary in their quality of content; they often provide inadequate or incorrect information."

Recommendation

Process: Bug reports should contain the elements most helpful for developers, which are (in order of helpfulness: steps to reproduce, stack traces, test cases, observed behaviour, screenshots, and expected behaviour.)

ITS: Provide a Bug Report template that must be used when submitting a Bug Report.

Context

Stakeholder Benefits: *Developers*: Get the information they need. *Reporters*: Get their reports resolved faster.

Stakeholder Costs: *Reporters*: Have to put more effort into submitting the required information.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: OSS where the reporters are external to the development organisation.

Exclusion Factors: None

Violation

Smells: Bug reports that consist of entirely unstructured text with no sections or headers.

Consequences: "Developers are slowed down by poorly written Bug Reports".

Causes: Reporters don't know what information to provide. Reports don't want to put in the time to provide all required elements. Reporters don't know how to get all the required elements.

Algorithmic Detection: Automate the detection of missing elements, and recommend that they be added before allowing the user to submit the Bug Report.

Extracted and extended from Zimmerman et al. [250] and Bettenburg et al. [27]

Table A.3.: BP02: Atomic Feature Requests.

Summary

Objective: Each Feature Request has only one request.

Motivation: "Users should be told to split Feature Requests into atomic parts: one request per issue ID. This makes it easier later on to follow up on the request and link other requests to it. When developers looking at the issue see that it is not atomic, they should close it and open two or more new ones that are atomic."

Recommendation

Process: Reporters should only submit one Feature Request per issue.

ITS: Unknown

Context

Stakeholder Benefits: *Issue Consumers*: It is easier to work with Feature Requests that are atomic. *Reporter*: Have confidence that their Feature Request will be better recieved because it is atomic.

Stakeholder Costs: *Reporter*: Must made Feature Requests atomic.

Artefact Scope: Issue

Issue Types: Feature requests (although it reasonably applies to other types such as Bug Reports).

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: None

Consequences: It can be difficult to work with Feature Requests that contain two parts: hard to discuss, how to associate the status, and how to assign it to multiple people. Non-atomic Feature Requests suffer traditional coupling dependency problems.

Causes: Inexperienced reporters don't know how to identify what an atomic "unit" looks like. Reporters think that that these "units" are related and therefore should be in a single report.

Algorithmic Detection: "Certain wordings can hint at non-atomic requests [...] such as 'a couple things'."

Extracted and extended from Heck and Zaidman [98]

Table A.4.: BP03: Assign Bugs to Individuals.

Summary

Objective: Achieve good Bug Report quality by having the correct fields set.

Motivation: "During the bug resolution process, a bug must be assigned to a particular developer so that it could be resolved. Whenever a bug is assigned to any team, and it is not specified which member of the team is going to solve that bug; it becomes everyone's problem but no one has its responsibility individually."

Recommendation

Process: Bugs should either not be allowed to be assigned to a team, or they should be considered an intermediate stage. Regardless, a bug should not be marked as "resolved" until it is assigned to an individual. A better process would be to have a separate custom field if assigning the bug to a team, and leave the real "assigned" field for individuals.

ITS: If an organisation wants to support team-based bug assignment, then they should instead create a custom field for "Assigned Team", and leave the regular assignee field to be used for individuals. Alternatively, don't allow bugs to be closed until the bug is assigned to an individual.

Context

Stakeholder Benefits: *Everyone*: Knows who was assigned to each bug.

Stakeholder Costs: *ITS Maintainer*: Cannot close a bug until someone is assigned.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: The organisation does not have a process in place in which bugs are first assigned to teams, who then decide which invidual within the team will address the bug.

Exclusion Factors: None

Violation

Smells: "Bugs assigned to a team."

Consequences: "Potential impacts of this process smell could be loss of traceability and accountability of bugs."

Causes: "Mistakenly assigned by the triager, Unavailability of a developer."

Algorithmic Detection: "First, we check whether the bug is assigned. If so, we search for the selected keywords: 'team', 'group' and 'backlog' in the assignee field. We found those keywords by manually inspecting the assignee names in each project. For example, in the MongoDB Core Server project history, there are some bugs assigned to Backlog-Sharding Team, which we consider it as a smell."

Extracted and extended from Qamar et al. [187, 188]

Table A.5.: BP04: Sufficient Description.

Summary

Objective: Provide a sufficiently long description.

Motivation: "In the issue tracking process, the description field plays a crucial role in providing a comprehensive explanation of the issue beyond the brief one-line summary. A well-written description can also help in identifying and resolving issues more effectively."

Recommendation

Process: "It is vital to ensure that the description has a sufficient length and provides a clear and comprehensive explanation of the issue."

ITS: Set a minimum-required length on the Description field. Also consider automating the checking of existing Descriptions for short ones that can be improved.

Context

Stakeholder Benefits: *Issue Consumers*: Benefit from more complete issues with more information. *Issue Creators*: Spend less time coming back to issues to add more information in the future when ultimately requested.

Stakeholder Costs: *Issue Creators*: Must write more information initially.

Artefact Scope: Issue

Issue Types: All

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: Blank descriptions.

Consequences: "This may lead to issues with short or even empty descriptions, which can make it difficult to fully understand the problem and its resolution. [...] According to research, the quality of the description is one of the most important factors in predicting reopened bugs. Bug tickets with insufficient descriptions are more likely to be reassigned, which is considered a problematic pattern in the ITS process."

Causes: Issue creators, particularly if they are outside the organisation, are likely unaware of how much tacit knowledge they have about their own bug, Feature Request, or support request. As a result, they believe a short title and description is sufficient to address their needs.

Algorithmic Detection: 1) Search for all issues with less than X characters/words. 2) Notify the appropriate stakeholders, e.g., the issue assignee, issue reporter, or perhaps a manager. 3) Alternatively, create a dashboard that visualises description lengths across issues, and leave it to a manager to decide which ones to act on.

Extracted and extended from Prediger [184]

Table A.6.: BP05: Succinct Description.

Summary

Objective: Issue descriptions should be succinct, particularly for requirements and development issues.

Motivation: Issue descriptions are the main descriptive tool for outlining the intention and purpose of an issue, whether that be a Bug Report, User Story, or work item. While some issue types (such as Bug Reports) benefit from as much information as possible, others (such as user stories and work items) are designed to communicate specific information quickly, in a structured or technical way.

Recommendation

Process: When writing descriptions, assess whether additional information is necessary. Add thoughts and opinions in the comments, to keep the description clean, clear, and to the point.

ITS: For certain issue types, it might be beneficial to automate the checking of description lengths before allowing submission of issues.

Context

Stakeholder Benefits: *Issue Consumers*: They will have an easier time consuming issues.

Stakeholder Costs: *Issue Creators*: Need to consider what is important (if not documented), and edit their issues to be more concise before submitting. *Issue Consumers*: May have to seek additional information that could have been added, but was not added due to the attempt at being more concise.

Artefact Scope: Issue

Issue Types: Most types except perhaps Bug Reports, which generally benefit from all known details.

Inclusion Factors: None

Exclusion Factors: If the orgnaisation uses the description field as a collection of all known information about the issue, then perhaps they will not want concise descriptions.

Violation

Smells: "Description too long." "Excessively long descriptions may indicate that the issue consists of multiple sub-tasks and should be broken down into smaller and more manageable parts."

Consequences: Issues can be hard to parse. Information can be lost within too much information. Large issues can cause information overload for people trying to interpret the issue. This can all lead to delays in addressing the issue, whether that be a Feature Request or User Story.

Causes: Issue creators might think that more information is better, regardless of the amount. Inexperienced issue creators might also not understand what is the minimal desired information, so they add as much as they can think of. Erroneous and non-helpful information can also be added by outsiders of the project when submitting Feature Requests and such.

Algorithmic Detection: If the organisation has defined constants for desired maximum length of certain issue types, then this can be used to automate the detection of such violations.

Extracted and extended from Lüders [137]

Table A.7.: BP06: Avoid Status Ping Pong.

Summary

Objective: Streamline issue state changes through the identification and removal of state cycles.

Motivation: Issues in ITSs utilise the status to denote what phase the issue is in. In cases where an external process disagreement exists, it can surface as cycles in the status of an issue. While it can be acceptable for a status to be reverted due to a mistake, or cycles to appear in certain company-defined processes, unwanted and unknown cycles occurring slows down the completion of tickets.

Recommendation

Process: If an unknown cycle is detected, discuss with the people involved why it is occurring and how to proceed without cycling again. If a known cycle is occurring, discuss with the people involved instead of using technology as a barrier to discussion.

ITS: Automate the detection of cycles and notify the involved people of detected cycles.

Context

Stakeholder Benefits: *Stakeholders*: Know that there are no unknown cycles occurring.

Stakeholder Costs: *Stakeholders*: Those involved with the cycles have to take the time to discuss together to resolve the underlying disagreement causing the cycling.

Artefact Scope: Issue

Issue Types: All

Inclusion Factors: "Large teams", "distributed teams", "FLOSS Projects".

Exclusion Factors: An organisation has defined a cycle within the process. E.g., the status change between "waiting for user response" and "waiting for developer response".

Violation

Smells: "[Change Requests] that are repeatedly resolved or reopened [...] ([...] "ping pong" as it was referred to by our informants) are worth looking into more deeply."

Consequences: Issues take much longer to resolve.

Causes: People don't realise there is a cycle occurring. Those involved with the cycle are abusing the mechanic to avoid other consequences (like having an issue assigned to them for too long without action).

Algorithmic Detection: 1) Create a state-transition graph from the history of all status changes in an ITS. 2) Note which status cycles are allowed (often none). 3) When an issue is assigned a new status, check the entire status history to record any cycles. 4) If the recorded cycles are not in the list of allowed cycles, then take some action. Actions include an automated comment on the issue noting the detect cycle activity, or a message sent to all members involved with the cycle to notify them that it is happening.

Extracted and extended from Halverson et al. [94], Aranda and Venolia [10], and Prediger [184]

Table A.8.: BP07: Avoid Assignee Ping Pong.

Summary

Objective: Streamline issue assignee changes through the identification and removal of assignee cycles.

Motivation: Issues in ITSs utilise the assign to denote who is in charge of the issue. In cases where an external process disagreement exists, it can surface as cycles in the asignee of an issue. While it can be acceptable for an assignee to be reverted due to a mistake, or cycles to appear in certain company-defined processes, unwanted and unknown cycles occurring slows down the completion of tickets.

Recommendation

Process: If an unknown cycle is detected, discuss with the people involved why it is occurring and how to proceed without cycling again. If a known cycle is occurring, discuss with the people involved instead of using technology as a barrier to discussion.

ITS: Automate the detection of cycles and notify the involved people of detected cycles.

Context

Stakeholder Benefits: *Stakeholders*: Those involved know that there are no unknown cycles occurring.

Stakeholder Costs: *Stakeholders*: Those involved with the cycles have to take the time to discuss together to resolve the underlying disagreement causing the cycling.

Artefact Scope: Issue

Issue Types: All

Inclusion Factors: "Large teams", "distributed teams", "FLOSS Projects".

Exclusion Factors: An organisation has defined a cycle within the process. E.g., the assignee change between developer and the tester as they find bugs (during testing) and fix them (during another round of development).

Violation

Smells: "[Change Requests] that are [...] repeatedly reassigned ([...] "ping pong" as it was referred to by our informants) are worth looking into more deeply."

Consequences: Issues take much longer to resolve.

Causes: People don't realise there is a cycle occurring. Those involved with the cycle are abusing the mechanic to avoid other consequences (like having an issue assigned to them for too long without action).

Algorithmic Detection: 1) Create a state-transition graph from the history of a all assignee changes in an ITS. 2) Note which assignee cycles are allowed (often none) 3) When an issue is assigned a new assignee, check the entire assignee history to record any cycles. 4) If the recorded cycles are not in the list of allowed cycles, then take some action. Actions include an automated comment on the issue noting the detect cycle activity, or a message sent to all members involved with the cycle to notify them that it is happening.

Extracted and extended from Halverson et al. [94], Aranda and Venolia [10], and Prediger [184]

Table A.9.: BP08: Set Bug Report Assignee.

Summary

Objective: Achieve good Bug Report quality by always setting an assignee.

Motivation: "Each reported bug must be triaged to decide if this bug describes a significant and new enhancement or problem, it must be assigned to an appropriate developer for further investigation. [...] However, it is observed that there are bugs that have no assignee at all even if the bug is resolved. This is a potential indicator that the BT process is not followed properly."

Recommendation

Process: When working on a bug, make sure you are the assignee. Before closing any bugs, make sure there is an assignee on the bug.

ITS: Make the "assignee" field mandatory, and don't allow "people" in Jira to be teams.

Context

Stakeholder Benefits: *Everyone*: Those who would benefit from traceability.

Stakeholder Costs: *ITS Maintainer*: Cannot close a bug until someone is assigned.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "It is observed that there are bugs that have no assignee at all even if the bug is resolved. This is a potential indicator that the [bug tracking] process is not followed properly. Thus, we consider it a process smell and call it an unassigned bug."

Consequences: "The potential impact of not assigning a developer to a bug could be a loss of traceability and delays in bug resolution."

Causes: "Developer's availability, Time pressure on triager, Triager could not find an expert developer."

Algorithmic Detection: "First, we check whether the bug is fixed and closed. If so, we check whether the assignee field is empty or not. However, there are also some cases where the assignee field is not empty but an invalid email address is written. For instance, if the unassigned@gcc.gnu.org address is used as an assignee email, we consider this case a smell as well."

Extracted and extended from Qamar et al. [187, 188] and Prediger [184]

Table A.10.: BP09: Set Bug Report Priority.

Summary

Objective: Achieve good Bug Report quality by always setting a priority.

Motivation: "Priority refers to how quickly a bug needs to be resolved and the order in which developers have to fix bugs. Correctly assigning bug priority is integral to successfully plan a software development life cycle."

Recommendation

Process: Assign a priority to every bug.

ITS: Have a separate initial stage for bugs called "triage", and don't allow bugs to preceed from this stage until this field is set.

Context

Stakeholder Benefits: *Managers*: Managing the backlog of bugs becomes easier.

Stakeholder Costs: *Developers*: Have to decide on priority.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "We are considering not prioritizing bugs as a process smell and calling it a missing priority."

Consequences: "The potential impact of this process smell is on the development-oriented decisions (time and resource allocation)."

Causes: "Expertise of triager, Wrong bug severity, Overlooked by triager."

Algorithmic Detection: "We check whether the priority field is valid or not. Some of the invalid priority strings are None, Not Evaluated, and "–"."

Extracted and extended from Qamar et al. [187, 188]

Table A.11.: BP10: Set Bug Report Severity.

Summary

Objective: Achieve good Bug Report quality by always setting a severity.

Motivation: "The Bug Report is triaged and the severity (e.g., low, medium, high) of the bug is assigned after a Bug Report has been submitted. The task of assigning a bug severity is a resource-intensive task. Severity ratings help in determining the priority of a bug i.e. in which order the bugs should be fixed. A bug could be incorrectly prioritized if the severity of the bug is not mentioned, which in turn can affect the quality of the product that is being developed."

Recommendation

Process: Assign a severity to every bug.

ITS: Have a separate initial stage for bugs called "triage", and don't allow bugs to preceed from this stage until this field is set.

Context

Stakeholder Benefits: *Managers*: Managing the backlog of bugs becomes easier.

Stakeholder Costs: *Triagers*: Have to decide on severity.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "If the severity information is missing, we consider it a process smell and call this smell missing severity."

Consequences: "It affects the resource allocation and planning of other bug fixing activities."

Causes: "Triager overlooks."

Algorithmic Detection: "We check whether the severity field is valid or not. Some of the invalid severity strings are N/A and "-". The detection mechanism may change across different BT tools. For example, Jira does not include a severity field by default but some organizations create their custom fields."

Extracted and extended from Qamar et al. [187, 188]

Table A.12.: BP11: Set Bug Report Environment.

Summary

Objective: Achieve good Bug Report quality by always setting the environment variables.

Motivation: "Every field in a Bug Report has its importance, and if they are present, they help the developer to resolve the bug quickly. A version of the product is an important field. If version information is missing in the Bug Report, the developer does not know in which version the user or tester is having this bug."

Recommendation

Process: Assign an environment to every bug.

ITS: Have a separate initial stage for bugs called "triage", and don't allow bugs to preceed from this stage until this field is set.

Context

Stakeholder Benefits: *Managers*: Managing the backlog of bugs becomes easier. *Developers*: Fixing bugs becomes easier.

Stakeholder Costs: *Developers*: Have to elicit (or find for themselves) the missing information.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Missing version information in the Bug Report is considered to be a smell."

Consequences: "The bug with no environment information is difficult to reproduce."

Causes: "Reporter forgets to mention, Reporter does not know the environment details, Reporter was short in time to mention information."

Algorithmic Detection: "We check whether the component, version, environment, and operating system fields are empty."

Extracted and extended from Qamar et al. [187, 188]

Table A.13.: BP12: Assignee Bug Resolution.

Summary

Objective: The bug assignee should be the one to resolve the bug.

Motivation: "In BTS, whenever a bug is encountered, it should be well documented and then resolved so that it can be traced later on if required. Therefore, it is important for traceability that bugs are assigned and resolved by the same person during their life cycle. To promote traceability, whoever is the assignee of a bug, that person should be the person to resolve the bug."

Recommendation

Process: Only allow the person who is assigned to the bug, be the one to resolve it.

ITS: Set a rule within the ITS that only the person currently assigned to it, is able to resolve it. Reject other inputs.

Context

Stakeholder Benefits: Unknown

Stakeholder Costs: Unknown

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Non-assignee resolver of bug."

Consequences: "It could be difficult to understand why some person has resolved a bug. [...] The potential impact of this process smell is on the traceability of a bug."

Causes: "Assignee forgot to close, Bug was originally resolved by someone else, Bug can be closed by administrative roles, Assignee might not be able to resolve and toss it to other developers."

Algorithmic Detection: "First, we check whether the bug is assigned and resolved. If so, we compare the assignee and the person who resolved the bug."

Extracted and extended from Qamar et al. [187, 188]

Table A.14.: BP13: Avoid Zombie Bugs.

Summary

Objective: Foster a meaningful backlog by keeping issues alive or resolving them.

Motivation: "Zombie bugs are defects that have lain dormant for a (relatively) long period of time. Low priority bugs that turn into zombies may not be a problem. However, being aware of zombie bugs was reported as an important part of project housekeeping."

Recommendation

Process: Regularly check how long bugs have been open, tending to those older than a certain time. Decide as an organisation how long bugs should remain open before they are set to "won't fix", or something similar.

ITS: Automate the resolution of bugs older than a certain amount of time. The resolution will be some kind of "won't fix", not the standard "fixed".

Context

Stakeholder Benefits: *Managers and Developers*: Have a meaningful backlog where all open bugs should be worked on.

Stakeholder Costs: *Managers and Developers*: Need to address these older bugs by either working on them, or changing the status to resolved. This cost is eliminated with the automated approach to marking bugs as resolved.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Defects [(Bugs)] that have lain dormant for a (relatively) long period of time."

Consequences: Backlogs grow indefinitely if not otherwise addressed or resolved, which leads to a lack of meaning to the status "open".

Causes: Developers forget about bugs.

Algorithmic Detection: 1) Search for all bugs older than X days (X defined by the organisation). 2) Either mark the bug as "won't fix" or "stale" as a resolution, or notify the assignee that they need to either act on the bug or change the status themselves.

Extracted and extended from Halverson et al. [94]

Table A.15.: BP14: Active Bug Reports.

Summary

Objective: Cultivate a meaningful open Bug Report backlog through regular activity or archiving.

Motivation: "Bugs that are left open for a long time or bugs that have incomplete resolution. In a [bug tracking system], once the bug is opened, it should not be left unattended or open for a long time. The knowledge of the bug may be forgotten over time. Even if it is not closed, some progress should be made to resolve the bug."

Recommendation

Process: Revisit Bug Reports that have not been updated in more than X months, and either perform a meaningful action towards resolution, or mark the Bug Report as "will not fix". The recommended maximum is 3 months, but a more reasonable limit (to maintain working knowledge) is 1 month.

ITS: Create a dashboard or notification system that contacts the relevant stakeholders when a Bug Report has not been updated in X months.

Context

Stakeholder Benefits: *ITS Users*: They know the open Bug Reports are meaningfully curated and represent live issues.

Stakeholder Costs: *Assignees*: Have to continuously assess whether it is worth keeping a Bug Report open or not.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: If the organisation maintains open bugs for future reference, i.e., a history of what was reported but never addressed.

Violation

Smells: Assignees ignoring issues and issues that are delayed for too long.

Consequences: "The potential impact of [the violation of this Best Practice is a] delay in the bug resolution process." Another potential impact is a growing backlog of open Bug Reports that may never be resolved.

Causes: "Incorrect severity indication, Inadequate bug description, Incorrect prioritization, Overlooked bug."

Algorithmic Detection: "We compare the dates of sequential activities in the bug history. While the bug is not resolved, if there is a gap longer than three months between any two activities, we consider it a [violation]."

Extracted and extended from Qamar et al. [187, 188] and Aranda and Venolia [10]

Table A.16.: BP15: Bug Report Discussion.

Summary

Objective: Achieve good Bug Report quality by encouraging discussion.

Motivation: "In BTS, comments can be posted by anyone in response to an initial Bug Report. Therefore, it means that for some notions of popularity, comment count can be used as a proxy. Textual contents of Bug Reports such as descriptions, summaries, and comments have been utilized by textual information analysis-based approaches to detect bug duplicates. In proposing the Bug Reopen predictor features like description features, comment features, and meta-features are being used. For identifying the blocking bugs, the most important features are comment size, comment text, reporter's experience, and the count of developers. Comments on the Bug Report serve as a forum for discussing the feature design alternatives and implementation details. Generally, the developers who have an interest in the project or who want to participate post the comments and indulge in a discussion on how to fix the bug."

Recommendation

Process: Unknown

ITS: Unknown

Context

Stakeholder Benefits: Unknown

Stakeholder Costs: Unknown

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: The organisation discusses Bug Reports in a different place, such as a chat tool or face-to-face.

Violation

Smells: "Considering the importance of comments in Bug Reports, we observed that some Bug Reports have no comments and consider it a process smell. We call this smell no comment bugs."

Consequences: "The potential impact of this process smell is on the bug resolution time and other linked software development activities such as developer recommendation, duplicate bug detection, and bug reopening prediction."

Causes: "Ignored bug, Developers/Contributors forget to write comments, Developer might be too busy to write a comment."

Algorithmic Detection: "First, we check whether the bug is closed. If so, we check whether there is at least one comment in the bug."

Extracted and extended from Qamar et al. [187, 188] and Tamburri et al. [218]

Table A.17.: BP16: Respectful Communication.

Summary

Objective: Maintain respectful discourse within issues and their comments.

Motivation: ITE bring many stakeholder groups together in one place, collaborating through a central tool and set of processes. These stakeholders include people external to the organisation, such as users of the software and others that are affected by its use. It can be difficult to keep everyone's perspective in mind while communicating about various topics, including support tickets, Bug Reports, Feature Requests, but it is important to maintain a respectful discourse.

Recommendation

Process: Maintain a respectful tone while interacting with people in an ITS, and remember that their experiences are likely drastically different than your own. Despite these differences, there are many interactions that are never okay, such as insults and threats.

ITS: Automated systems can be put in place to check the textual content within issues, including the summary, description, and comments. These solutions utilise NLP to detect things like negative sentiment, bad words, and hate speech.

Context

Stakeholder Benefits: *Everyone*: Benefits from a respectful communication environment.

Stakeholder Costs: *ITS Maintainer*: Needs to implement such a system. *Managers or Moderators*: Need to regularly review flagged content and moderate the discussions.

Artefact Scope: Issue

Issue Types: All

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Toxic discussions."

Consequences: Developers can become frustrated and disconnected from the community they are developing for. Communities can feel distant from the orgnisation, and ultimately leave for competing products. A bad image and reputation of an organisation can grow from small negative interactions within the ITS.

Causes: Lack of empathy for users, developers, and managers. Miscommunication. Different styles of discussion misaligning. Lack of issue discussion moderation.

Algorithmic Detection: Apply NLP techniques to detect negative language, and then handle appropriately.

Extracted and extended from Lüders [137]

Table A.18.: BP17: Consistent Properties.

Summary

Objective: Issue properties should always represent the most up-to-date information.

Motivation: Issues are both a form of structured information, and unstructured dialogue. This means that it is possible, and indeed often the case, that information in the description or comments contradicts information listed in the properties. For example, someone comments that this issue is a duplicate of another issue, but does not add this as a "duplicated by" link in the properties. Over time, this leads to inconsistencies between the properties and the dialogue.

Recommendation

Process: Update properties instead of updating the description or adding a comment.

ITS: Automate the detection of properties in the description and comments, and warn assignees and managers about the existence of such properties. This can be done with some certainty, but natural language of course makes this a task that comes with uncertainty.

Context

Stakeholder Benefits: *Everyone*: Can trust the properties as being up-to-date.

Stakeholder Costs: *Everyone*: Must actually update the properties. *Privileged Users*: Must update properties after non-privileged users add comments with new information.

Artefact Scope: Issue

Issue Types: All

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: Properties are mentioned in the comments or description.

Consequences: The entire issue must be read in order to understand the current state, and ultimately to trust the properties as they are set. While this is possible for any single issue, this is not manageable for the entire ITS.

Causes: People may not have the permission to update the issue itself, so they make a comment instead. People may not know that there is a property for this information. People might be lazy, and prefer to make a quick comment.

Algorithmic Detection: Search the text for potential property updates. This includes searching for the names of the properties, as well as all states the properties can be in. For example, search for the word "status", but also search for "Open" and "Resolved". With some additional NLP intelligence, the false positives can be reduced to a manageable low number.

Extracted and extended from Lüders [137]

Table A.19.: BP18: Good First Assignee.

Summary

Objective: Achieve good Bug Report quality by assigning the best developer first.

Motivation: "Research shows that the time-to-correction for a bug is increased by the reassignments of developers to a bug".

Recommendation

Process: Unknown

ITS: Unknown

Context

Stakeholder Benefits: Unknown

Stakeholder Costs: Unknown

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Reassignment of bug assignee."

Consequences: "The potential impact of this process smell is an increase in the bug fixing time, which eventually delays the delivery of the product."

Causes: "Reassignment of some fields cause others to be reassigned, Triager does not know the suitable developer, The developer recommendation system is not integrated into BTS, Admin batch operations."

Algorithmic Detection: "The mining strategy for this smell is to look in the bug history for the assignee property. If the assignee field is changed more than twice, then we count it as a smell. Also, we observed that there are some multiple assignee changes done in a very short interval. We consider these multiple assignee changes as a mistake and do not count them as a smell. The interval duration is set as five minutes. Therefore, if multiple assignments are done in five minutes, they are counted as one assignment."

Extracted and extended from Qamar et al. [187, 188]

Table A.20.: BP19: Stable Closed State.

Summary

Objective: Increase the confidence that once a bug is closed, it will stay closed

Motivation: "Reopened bugs are those that were previously closed by the developers but were later reopened for various reasons (such as not being reproduced by the developer or improperly tested fix). Reopened bugs reduce the overall software quality, increase maintenance expenses, as well as unnecessary rework by developers."

Recommendation

Process: Unknown

ITS: Unknown

Context

Stakeholder Benefits: Unknown

Stakeholder Costs: Unknown

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "In a project when a significant number of bugs are reopened frequently, then it can be an indication that the project is already in trouble, and may be heading towards trouble soon. We call this smell closed-reopen bug ping pong; as it is the ping pong among the bug states during its life cycle."

Consequences: "Potential impacts of reopened bugs could be; they take a notably longer time to resolve. Reopened bugs also increase development costs, affect the quality of product, prediction of release dates reduce the team morale leading to poor productivity."

Causes: "Insufficient unit testing, Ambiguous bug specifications, Changed bug scope, Poorly/incorrectly fixed bugs, Tester not testing properly."

Algorithmic Detection: "We check the history of the status field of the bug. Some projects explicitly use REOPENED value for the status field while others do not. In such cases, we check whether the status field is changed from Closed to another value, and we count it as a smell. Also, we observed that there are some multiple status changes in a very short interval. We consider these changes a mistake and do not count them as a smell. The interval duration is set as five minutes. Therefore, if a bug status is changed multiple times in five minutes, it is counted as one change."

Extracted and extended from Qamar et al. [187, 188]

Table A.21.: BP20: Timely Severe Issue Resolution.

Summary

Objective: Address severe issues within a defined timeframe.

Motivation: While issue resolution time is based on many factors, a high severity should mean a timely resolution. A high severity is often defined as some kind of blocker for the third-party involved, such that their business is impacted and waiting for a resolution to this issue. Therefore, high severity issues should be resolved with a small organisation-defined time window. For example, severity 2 issues could have a defined maximum resolution time of 48 hours.

Recommendation

Process: Agree on resolution maximums for higher severity issues, and review high severity issues daily to check for progress.

ITS: Create a dashboard for viewing all high-severity issues, sorted on time open (descending). Create a notification system for the assignee such that they are notified every X hours that this ticket is still open. Perhaps a morning notification to help structure their day around addressing this issue.

Context

Stakeholder Benefits: *Problem-affected Users*: Get their issues resolved within a known time frame.

Stakeholder Costs: *Managers or Developers*: Have to keep an eye on the list of high severity tickets. *Developers*: Must address issues under time pressure.

Artefact Scope: Issue

Issue Types: Bug and other problem-related types such as "Security Flaw"

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: High-severity issues are left unresolved for long periods of time.

Consequences: Impacted organisations are left with serious unsolved issues. Terms of Use could be violated if promises were made regarding support response time.

Causes: Team is not aware of a high severity issue. People are waiting on others to get back to them.

Algorithmic Detection: 1) Search for issues of a certain severity that have been open for X hours/days (where X is defined by the organisation). 2) Notify the involved people that progress needs to be made.

Extracted and extended from Halverson et al. [94] and Prediger [184]

Table A.22.: BP21: Issue Creation Guidelines.

Summary

Objective: Guide users how to create good issues.

Motivation: Organisations should communicate their specific requests for how they want issues of different types filled in. With such guidelines, users can be directed to them before submitting issues.

Recommendation

Process: "Projects should include a link to clear guidelines on how to enter issues (e.g. when is it a defect or an enhancement) to ensure that all fields are filled correctly and to avoid users entering new requests for new versions of the software."

ITS: None

Context

Stakeholder Benefits: *Internal Stakeholders*: Get issues submitted that are more complete. *External Stakeholders*: Have more confidence that their issues will be accepted since they have followed specific instructions on how to submit issues of certain kinds.

Stakeholder Costs: *External Stakeholders*: Have to read and follow these guidelines.

Artefact Scope: Issue

Issue Types: All

Inclusion Factors: Organisations that have specific desires for how their issues are submitted.

Exclusion Factors: None

Violation

Smells: Organisations that don't have issue creation guidelines.

Consequences: Issues are created that are missing important aspects.

Causes: People are unaware of what is needed from them.

Algorithmic Detection: Unknown

Extracted and extended from Heck and Zaidman [98]

Table A.23.: BP22: On-Topic Discussions.

Summary

Objective: Maintain topic-related discussions on Feature Requests.

Motivation: ITS contain a lot of information, and it can be difficult to consume and understand it all. This is exaggerated by issue descriptions and issue comments that are off-topic. There are other places to have such discussions, such as forums.

Recommendation

Process: "Users entering new Feature Requests should only include issue-related comments; the same holds for the users commenting on existing Feature Requests. For other types of comments the mailing/discussion list should be used (from where the user can easily hyper-link to the request). Projects could even go as far as removing unrelated comments from the request, to keep a 'clean' database."

ITS: Show a "hint" message while users are entering comments that off-topic comments will be deleted.

Context

Stakeholder Benefits: *Issue Consumers*: Hae an easier time finding the relevant information, since it is all on-topic.

Stakeholder Costs: *Commenters*: Need to refrain from making off-topic comments on issues, and find another place for them.

Artefact Scope: Issue

Issue Types: Feature requests (although it reasonably applies to all other types).

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: Comments on Feature Requests are discussing off-topic things.

Consequences: Issues become hard to parse. Information that is meant for somewhere else is lost in this issue.

Causes: Commenters are unaware that their discussions are off topic. Commenters don't know where else to voice these discussions.

Algorithmic Detection: Unknown

Extracted and extended from Heck and Zaidman [98]

Table A.24.: BP23: Bug-to-Commit Linking.

Summary

Objective: Achieve good Bug Report quality by fostering traceability between bugs and resolving commits.

Motivation: "To obtain useful information about a software project's evolution and history." Bugs are resolved by committing code that resolves the reported problem. If a bug is revisted in the future, commit linking allows them to quickly navigate to the correct piece of code. Additionally, these links can be used in reverse to understand why a commit was conducted, either for reasons of insufficient commit messages, or building automated support for forwards and backwards traceability.

Recommendation

Process: "[...] all bug-fixing commits should be linked to their respective Bug Reports." Before a bug is marked as "resolved" and therefore closed, the associated code commit should be conducted, and the link to that commit should be pasted into the associated Bug Report.

ITS: Have a dedicated "Commit" field that is mandatory before the Bug Report can be marked as "Resolved".

Context

Stakeholder Benefits: *Developers*: When revisiting bugs to learn or trace what happened, they will know what code was implemented. *Testers*: They know what code was added based on a Bug Report.

Stakeholder Costs: *Assignee*: Has to add the link to the commit in the Bug Report.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: The organisation chooses not to link their ITS with GIT.

Violation

Smells: "If a bug is closed without any link to the bug-fixing commit, then in the future, it will be difficult to discover what happened with that bug. We evaluate this lack of traceability as a process smell and we call this No Link to Bug-Fixing."

Consequences: "The potential impact of the [violation of this Best Practice] is losing track of the bugs and eventually the traceability of the bug decreases. It also affects the related software development tasks such as prediction of bug locations, recommendation of bug fixes, and software cost."

Causes: "Developer forgets to mention, Committing link is not straightforward in [ITS], Weak understanding of [ITS]"

Algorithmic Detection: "First, we check whether the bug is fixed. If so, we check the comments and designated fields to find a link to the version control system."

Extracted and extended from Qamar et al. [187, 188]

Table A.25.: BP24: Link Duplicates.

Summary

Objective: Maintain good Bug Report quality by referencing related closed bugs.

Motivation: "If the problem of duplicate bug recognition is solved, it enables the developers to fix bugs more efficiently rather than waste time resolving the same bug. However, it is observed that some bugs that are marked as duplicates in BTS are not referenced to the original bug within the references section of a Bug Report. Instead, the reporters only put the duplicate keyword into the status section, which reduces the traceability."

Recommendation

Process: Always include a link to the duplicate bug when referencing it in another bug.

ITS: Have a separate "duplicated by" link type and use that when referencing bug duplicates.

Context

Stakeholder Benefits: *Developers*: Can easily find the "master" duplicate report when searching issues. *Outsiders*: Can more easily find duplicates during their search for related issues.

Stakeholder Costs: *Maintainers*: Have to add the duplicate link every time they mention a duplicate issue.

Artefact Scope: ITS

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "As far as we have observed, most of these bugs still reference the duplicate bug in some way, such as referring to it in the comment section. But some of them are marked as duplicate and do not have any reference to the duplicate bug, and it is a deviation from ideal [bug tracking] behavior. Therefore, we call it not referenced duplicates smell."

Consequences: "The potential impact of this process smell is on the identification of master Bug Reports."

Causes: "Person marking duplicate is new to the system, Being unaware of the previous bugs, Poor search feature of BTS to find duplicates."

Algorithmic Detection: "First, we check whether the bug is marked as a duplicate. If so, we check the linked issues field to find whether the duplicate bug is linked and has a reference to the other bug."

Extracted and extended from Qamar et al. [187, 188]

Table A.26.: BP25: Minimal Link Types.

Summary

Objective: Reduce redundant link types to simplify linking.

Motivation: ITSs often support linking between issues, to describe relationships between them, such as "duplicated by" and "is contained by". As organisations and their ITSs grow, the number of relationship types can grow as well, resulting in many link types. "The potential risk is that stakeholders cannot differentiate the link types from each other, such as Depend and Block, and be unsure which to choose."

Recommendation

Process: Organisations should manage the number of link types they add over time, trying to maintain a minimal set.

ITS: None

Context

Stakeholder Benefits: *Everyone*: Easier to understand and utilise linking process.

Stakeholder Costs: *ITS Maintainer*: Must routinely assess (and discuss with others) the need for multiple similar link types, making trade-offs between descriptiveness and conciseness of the link types.

Artefact Scope: ITS

Issue Types: All

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Too many link types."

Consequences: People can be confused about the different issue types, and "be unsure which to choose". "This can potentially lead to stakeholders not using a specific link type (Relate) or not linking at all."

Causes: Organisational growth and multiple independent processes being supported by the same ITS can lead to such similar link types. Once there are (too) many, it then can be the case that people simply don't know how to use them all.

Algorithmic Detection: None

Extracted and extended from Lüders [137]

Table A.27.: BP26: Record Links.

Summary

Objective: Record all issue links using the linking feature, not in comments or in the description.

Motivation: "If a link is known but only recorded in the comments, it is only visible from one issue report. E.g., issue A is related to issue B. However, it is not recorded and only commented on in issue A. If a stakeholder is viewing issue B, they are unaware of either the link to issue A or issue A. However, issue A might contain helpful information to resolve issue B. Furthermore, if issue B is supposed to be resolved in the following software cycle and issue A is a blocker of issue A, issue B might have to be delayed for the following software cycle."

Recommendation

Process: Record issue links using the official feature, not using natural language in the description or comments.

ITS: Automate the detection of links in the description and comments, and recommend that they are added as official links instead.

Context

Stakeholder Benefits: *Everyone*: Benefits from less hidden dependencies.

Stakeholder Costs: *Link Reporters*: Need to use the official linking feature, which might take longer than a quick comment.

Artefact Scope: ITS

Issue Types: All

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Known link was not recorded."

Consequences: Relationships between issues can be hidden if only mentioned in one of the linked issues. This can lead to hidden dependencies that delay or completely stop progress on issues, or lead to duplicate work.

Causes: People don't have the right permissions to add links themselves, but they have the permissions to post comments. People don't want to go through the effort of making an official link.

Algorithmic Detection: Links in ITSs have special formatting that can be detected in natural language comments. Once detected, certain people can be notified, whether that is the assignee of the issue, or a manager of the ITS that can then act on that information.

Extracted and extended from Lüders [137]

Table A.28.: BP27: Realistic Dependencies.

Summary

Objective: Catch and resolve unrealistic dependencies in link networks such as cycles.

Motivation: Issue links are formed naturally, one at a time, over the lifetime of an ITS. Small mistakes in this natural process can lead to unrealistic dependencies, such as a circular loop of "blocked by", such that no issue can be worked on because they all block each other in a loop. Without proper automated systems, these can be very hard to catch manually.

Recommendation

Process: None

ITS: Automate the detection of these unrealistic dependencies.

Context

Stakeholder Benefits: *Everyone*: Benefits from these hidden unrealistic dependencies being resolved.

Stakeholder Costs: *Assignees and Managers*: Have to resolve these dependency issues.

Artefact Scope: ITS

Issue Types: All

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Circular dependencies."

Consequences: Issue resolution can be delayed or completely blocked.

Causes: Mistakes in link assignment. Lack of experience with linking issues. Misunderstanding and misusing link types.

Algorithmic Detection: 1) Define what dependency arrangements are unrealistic, e.g., circular dependencies. 2) Form dependency graphs from the issues and links. 3) Notify the involved people of the unrealistic dependecies. This could be the assignees of the issues, or a manager across a larger part of the system.

Extracted and extended from Lüders [137]

Table A.29.: BP28: Singular Relationships.

Summary

Objective: Simplify ITS dependencies by limiting 1-to-1 relationships to a single link.

Motivation: "If two issue reports are linked multiple times, it can simply be noise. However, suppose the link type is conflicting, such as Duplicate and Block. In that case, the developer has to read and understand both issues to figure out the correct relation, which is unnecessary overhead."

Recommendation

Process: Don't allow two issues to have more than one relationship between them. Pick the link that best represents the relationship (often the more specific one). For example, the link type "is blocked by" can also be paired with the link type "depends on", but the latter is less specific and therefore has less meaning and usefulness.

ITS: Automate the detection and notification of multi-link pairs.

Context

Stakeholder Benefits: *Everyone*: Less confusion when approaching issues.

Stakeholder Costs: *Everyone*: Must pick the best link to represent a complex relationship.

Artefact Scope: Pairs of Issues

Issue Types: All

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Issues with multiple links."

Consequences: Can lead to increased confusion when working with the issues.

Causes: Confusion regarding which link types to use.

Algorithmic Detection: 1) Check for pairs of issues with more than one link between them. 2) Notify the assignee (or manager) about the problem.

Extracted and extended from Lüders [137]

Table A.30.: BP29: Connected Hierarchies.

Summary

Objective: Connect hierachies with links to view the full picture.

Motivation: Some issue types naturally form hierarchies when connected with their related issue types. For example, Epics should be linked with User Stories, which are then linked with Work Items. An Epic with no links—and therefore no hierarchy—is not yet part of the complete structure it was meant to form. This could be because it is "an idea that is not yet planned", but it could also be that the person configuring the hierachy "selected the wrong issue" on accident.

Recommendation

Process: All hierarchical issue types should be linked, or a separate system (such as tags) should be used to explicitly state that there needs to be a link in the future.

ITS: When creating hierarchical issues, the system could remind the creators (or assignees) some time later if there is still no link in the system. For certain types, such as User Stories, it could be configured to be mandatory that it is linked to an Epic before it is allowed to be created.

Context

Stakeholder Benefits: *Issue Consumers*: Get the full picture when working with hierarchical issues. *Managers*: Have a better understanding and view of the system when organising projects.

Stakeholder Costs: *Issue Creators*: Need to set the appropriate issue and link type and link to the correct issues.

Artefact Scope: ITS

Issue Types: Hierarchical Issue Types such as Epics, User Stories, and Work Items

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Isolated Epics. Isolated Subtasks."

Consequences: "It might get lost in the backlog", "a duplicate epic might be created later", "Subtasks without links to a parent-issue report have no link to the big picture". "Not knowing which Feature Request, User Story, or other issue report type a subtask belongs to can potentially impact the quality of the parent issue report to be resolved, as developers might overlook the subtask unless explicitly assigned."

Causes: Creator could be unaware of the need or importance of hierarchical links. The wrong issue or link type could have been selected.

Algorithmic Detection: 1) Define which hierarchical link types need to be applied to which issue types. 2) Check for issues that should have those link types but don't, as well as link types that should be connected to those issue types but are not. 3) Notify the responsible stakeholders (e.g., creator, assignee, manager) and ask them to act on this information.

Extracted and extended from Lüders [137]

Table A.31.: BP30: High-Dependency Bugs First.

Summary

Objective: Foster a low-dependency ITS.

Motivation: ITSs are composed of complex networks of dependencies between issues, such as "is blocking". These often form invisible barriers to addressing the work, and so it is important to increase the priority on bugs that are blocking other work, such as other bugs or even features.

Recommendation

Process: Prioritise and address bugs that block other issues first. Regularly check for bugs that are blocking others, and increase their priority to encourage devs to work on them.

ITS: Automate the detection of these dependencies and either notify people or automatically raise the priority. Visualise these dependencies so people can manually identify and make decisions based on the dependencies.

Context

Stakeholder Benefits: *Managers*: Have increased confidence that their ITS is clean of blocking dependencies. *Developers*: They know that awareness will be drawn to dependency-rich bugs, thus they don't have to worry or check manually all the time to find them themselves.

Stakeholder Costs: *ITS Configurer*: Has to implement the detection solution.

Artefact Scope: Issue

Issue Types: Bug

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: "Bugs that block too much."

Consequences: Not addressing a high-dependency blocking bug can lead to people or processes waiting indefinitely.

Causes: People are unaware of the blocking bug. People don't work on the bug because it is too low priority. The importance of the issues being blocked is not made apparent to the person assigned to the blocking bug.

Algorithmic Detection: 1) Search for dependencies by constructing graphs. 2) Increase awareness towards this blocking bug in a number of ways: a) Notify the assignee that their bug is blocking others, b) increase the priority automatically, c) visualise this graph for others to make decisions.

Extracted and extended from Halverson et al. [94]

Table A.32.: BP31: Search Reminders.

Summary

Objective: Decrease the number of duplicate Feature Requests by prompting users to search for existing ones first.

Motivation: "Research has shown that most duplicates are reported by infrequent users so giving them a reminder of the procedure or explicit instructions could help filter out some of the duplicates."

Recommendation

Process: "Projects should include warnings to search first and to ask on mailing or discussion lists before entering a new request."

ITS: Prompt the user to search for existing Feature Requests before they open a new Feature Request. As the user is writing their Feature Request, show them existing Feature Requests that overlap conceptually with theirs (using NLP techniques).

Context

Stakeholder Benefits: *Developers*: Increased confidence that new Feature Requests are novel. *Reporters*: More acceptance of their reports, since they are novel.

Stakeholder Costs: *Reporters*: Have to spend time searching for existing reports with similar keywords and concepts.

Artefact Scope: ITS

Issue Types: Feature Requests, but also applies to Bug Reports.

Inclusion Factors: None

Exclusion Factors: None

Violation

Smells: None

Consequences: Users enter duplicate Feature Requests.

Causes: Users are unaware of existing Feature Requests.

Algorithmic Detection: There are many algorithms design to detect duplicate issues, including Bug Reports and Feature Requests.

Extracted and extended from Heck and Zaidman [98]

Table A.33.: BP32: Ordered Product Backlog.

Summary

Objective: Maintain an ordered product backlog for meaningful product direction.

Motivation: In the Scrum methodology, the product backlog serves as the primary product direction, and therefore should be well-maintained and ordered. There are many backlog ordering techniques, of which multiple are often used at once. It is not important which technique is used, but rather that such orderings exist and support the process of maintaining a strong product direction.

Recommendation

Process: "Organize Product Backlog grooming meetings every other week where the top of the Product Backlog is ordered according to the value of items and taking care of the dependencies between work items."

ITS: Utilise the Severity, Priority, or other custom fields to differentiate the ITS items in meaningful ways.

Context

Stakeholder Benefits: *Stakeholders*: Benefit from a stronger product. *Customer*: Benefits from having a product better suited to their needs. *Product Owner*: Benefits from being supported by the ordered product backlog.

Stakeholder Costs: *Product Owner*: Must regularly update the backlog items' properties to keep them up-to-date and meaningful to the current product direction. *Customer*: Needs to provide regular feedback so the backlog can be adjusted accordingly.

Artefact Scope: Project/Product

Issue Types: User Stories (primarily), and other RE types.

Inclusion Factors: Team uses Scrum practices.

Exclusion Factors: "Projects where the requirements are fixed and given up-front by the customer, and these requirements are not going to change. Some initiatives by governments and military might have such requirements."

Violation

Smells: "Unordered product backlog." "The Product Backlog is not ordered, but Teams select items based on their own judgment."

Consequences: "As Product Backlog is unordered, the Team might be lacking vision of the risky or valuable elements of the product. As a result the Team might be building wrong features which do not have value to the customer or are just rarely used. Only features which are fun to implement get implemented as the Team starts to pick whatever they like from the backlog. Features which are hard to implement or test are left to the backlog and implemented last. This increases the risk of problems arising in the late stages of development."

Causes: "Product Owner without authority anti-pattern." "Customer Product Owner anti-pattern." "Long or non-existent feedback loops anti-pattern." "Insufficient competence of the Product Owner."

Algorithmic Detection: 1) Define which fields are important to the ordering of the product backlog. 2) Check all open issues to find issues where these fields are blank. 3) Notify the appropriate stakeholders of the importance of these fields and list which ones are currently blank.

Extracted and extended from Eloranta et al. [53, 54]

Table A.34.: BP33: Team-Produced Work Estimates.

Summary

Objective: Create accountability between teams and their tasks by allowing them to make their own work estimates.

Motivation: Teams that produce their own work estimates are said to be more connected to their work, and feel more accountable to those estimates. However, it is not uncommon to have product owners produce the estimates instead, thus potentially creating a disconnect.

Recommendation

Process: "Teams should produce work estimates for themselves. Try out planning poker. In the beginning, it might feel uncomfortable but just keep going and it will start to make more sense. After a while, you can ask the Team to do the estimates even without poker."

ITS: Utilise a "work estimate" field, and only allow the assigned person to enter the hours (following the team meeting)

Context

Stakeholder Benefits: *Developers*: Feel more connected to the work items. Also more likely to complete on time, given that they made the estimate.

Stakeholder Costs: *Manager*: Story points have to be estimated and modified over time, to adapt to the team's abilities. *Teams*: Must estimate their work items.

Artefact Scope: Project/Team

Issue Types: All

Inclusion Factors: Team uses Scrum practices.

Exclusion Factors: None

Violation

Smells: "Work estimates given to teams." "A product manager or the Product Owner produces work estimates."

Consequences: "The anti-pattern results in Teams that lack commitment: if there is slack time in Sprint, Teams do not pull features from the Product Backlog into the Sprint. On the other hand, given unrealistic Sprint goals may demotivate the Team and result in poor performance. Wrong estimates: the Team is better in estimating the effort than a single person who is not going to implement the task. Wrong estimates may lead to too optimistic or too pessimistic schedules."

Causes: "Hierarchical working practices of the company. Need to know an estimate in advance on how much the product will cost."

Algorithmic Detection: Check for the lack of a "work estimate" custom field, and check who is entering this number.

Extracted and extended from Eloranta et al. [53, 54]

Table A.35.: BP34: Story Points Over Hours.

Summary

Objective: Manage expectations and work towards correct predictions of work effort required per sprint.

Motivation: "The number of items selected from the Product Backlog for a Sprint is solely up to the Team. In earlier Scrum descriptions it was advised to use story points for estimating how many user stories can be taken into a Sprint."

Recommendation

Process: "Use story points to estimate stories. In this way, the estimates are based on relative sizing. In the first Sprint guess how many points you can deliver and after that use velocity to estimate how much can be done within a Sprint. Story points make sure that nobody expects a certain feature on a certain date."

ITS: Use a "story points" field, not an "hours" field.

Context

Stakeholder Benefits: *Developers*: Don't feel burdened by an actual hour amount on the issue.

Stakeholder Costs: *Manager*: Story points have to be estimated and modified over time, to adapt to the team's abilities.

Artefact Scope: Project/Team

Issue Types: All

Inclusion Factors: Team uses Scrum practices.

Exclusion Factors: None

Violation

Smells: "Hours in progress monitoring." "All user stories are estimated using hours."

Consequences: "As a consequence of using hours in progress monitoring, estimates can become inaccurate and they are often exceeded. A lot of time is wasted estimating how much time is required for the stories and even then the estimate is often inaccurate. User stories that require studying how to implement something are hard to estimate."

Causes: "Hierarchical organization where managers need to report how much work in hours is still left. Hours are used to calculate the price of the project and/or feature and estimates are used to find out the estimated price of a feature."

Algorithmic Detection: Check for an "hours" field in the ITS. Could also check for story points and make sure they are assigned for all items in a sprint.

Extracted and extended from Eloranta et al. [53, 54]

Table A.36.: BP35: Estimate all Items.

Summary

Objective: Estimate all work items before starting a sprint.

Motivation: "The scope and duration of the iterations in an agile project are typically defined by the development team that must commit to the iteration goals and deadlines." It is important for teams to estimate the amount of effort needed on sprint items before the sprint begins, otherwise there is no understanding of the actual work being agreed on for the sprint.

Recommendation

Process: Estimate all issues in a sprint before that sprint begins.

ITS: Use an issue field to hold the estimate for the issue. Block the issue from going into a "planned sprint" until it has an estimate. Send alerts to people who are assigned tickets when their issue doesn't have an estimate.

Context

Stakeholder Benefits: *Customer*: Will get software delivered more consistently. *Developers*: Feel more confident that they can finish the sprint items.

Stakeholder Costs: *Developers*: Need to estimate each issue before the sprint starts.

Artefact Scope: Issue

Issue Types: Work Items

Inclusion Factors: Team uses Agile. Team uses estimates.

Exclusion Factors: None

Violation

Smells: Issues are missing an estimate.

Consequences: "The development team could be committed to a deadline without a good understanding of the effort to deliver the iteration scope." Issues are not done in time for the sprint. Customer is unhappy about promised work not being finished on time.

Causes: Teams don't understand the importance of estimates.

Algorithmic Detection: 1) Find issues in current sprints that are missing an estimate. 2) Notify the assignees of the missing estimates. 3) Notify the manager of the missing estimates.

Extracted and extended from Telemaco et al. [221, 222]

Table A.37.: BP36: Avoid Unplanned Work.

Summary

Objective: Focus on planned work during a sprint, to achieve the agreed commitment.

Motivation: "Agile teams usually commit to delivering a set of features before an iteration begins. To achieve the agreed commitment, the teams must work without interference, following the iteration plan and unplanned work should be avoided."

Recommendation

Process: Only address planned work, and push unplanned work to the next sprint.

ITS: Once a sprint has started, either don't allow additional work to be added (lock that feature in the ITS), or automatically tag all unplanned work added to the ongoing sprint as "unplanned", which then highlights the negative nature of this decision.

Context

Stakeholder Benefits: *Managers*: Can confidently refuse to add unplanned work to the sprint because it is a rule that has been decided by the organisation. *Developers*: Gain confidence in the process that the agreed-upon workload for a sprint will remain that way.

Stakeholder Costs: *Managers*: Must refuse to add additional work to the sprint. *Customers*: Must become comfortable with waiting for features.

Artefact Scope: Issues in a Sprint

Issue Types: All

Inclusion Factors: Team uses Agile.

Exclusion Factors: Team is comfortable with unplanned work.

Violation

Smells: "The Unplanned Work Work smell smell is is detected when when tasks tasks are are included in in a a given iteration after it it starts. The presence of of this smell may indicate the unplanned tasks are jeopardizing the commitment with the iteration deadline."

Consequences: "Unplanned tasks [can jeopardize] the commitment with the iteraction deadline." Unplanned work can add undue stress to the sprint process. Trust in the agile process can degrade as sprint plans are repeatedly abused.

Causes: Demanding customer gets what they want. Product owners (managers) are not strict enough with refusing unplanned work. Serious problems such as severe bugs and security flaws are discovered that need to be addressed. If the team is already relaxed about pushing unfinished work into the next sprint, then taking on unplanned work becomes easier and more likely to happen.

Algorithmic Detection: Detect when issues are assigned to a sprint after the sprint has begun, and notify the appropriate stakeholders.

Extracted and extended from Telemaco et al. [221, 222]

Table A.38.: BP37: Recommended Sprint Length.

Summary

Objective: Maintain a core element of Scrum by using 2–4 week sprints.

Motivation: When adopting agile processes, it is tempting to modify certain elements to match what is already conducted within the company, to not have to adapt too many things. However, sprint length is one of the core elements of Scrum, and should be followed to ensure the benefits of Scrum are realised.

Recommendation

Process: Experiment with different Sprint lengths to find out which is a suitable Sprint length in the project. Start with a longer Sprint, e.g. four weeks and use that for a couple of months. Then try a shorter Sprint length. If the new length feels too short, go back to a bit longer Sprint length. One could try as short as 1.5 weeks length although that might cause too much overhead in the form of Sprint reviews and Sprint planning.

ITS: Keep track of the sprint lengths in the ITS, and graph them over time to show progress (or not) towards this essential Scrum process.

Context

Stakeholder Benefits: *Managers*: Benefit from seeing the progress towards proper Scrum processes.

Stakeholder Costs: *Managers*: Have to review and convince the team to adhere to certain sprint lengths.

Artefact Scope: Project/Sprint

Issue Types: All

Inclusion Factors: Team uses Agile (Scrum).

Exclusion Factors: "If the development must be synchronized with external work that has slower pace (e.g. hardware development), longer Sprints may be justified."

Violation

Smells: "Too long sprint." "Using 4 weeks or even longer Sprints."

Consequences: "In spite of longer working time for Sprints, the planned tasks tend to be unfinished at the end of a Sprint, possibly because the first weeks are not used efficiently enough and too large tasks are allocated to the Sprints. Too long sprints may prevent the customer from getting a required feature in time as priorities might change rapidly. Something more valuable can come up during the Sprint and this may lead to Customer Caused Disruption. As Sprints are long, more disruptions might occur and predictability is not so good anymore."

Causes: "Early experimenting with Scrum, waterfall legacy of long production cycles, customers unwilling to participate in short intervals."

Algorithmic Detection: 1) Calculate the length of the sprints from the sprints defined within the ITS. 2) Report those to the appropriate stakeholder or visualise them in a dashboard for stakeholders to review.

Extracted and extended from Eloranta et al. [53, 54]

Table A.39.: BP38: Consistent Sprint Length.

Summary

Objective: Maintain constant and simplified velocity by using the same sprint length for every sprint.

Motivation: When adopting agile processes, it is tempting to modify certain elements to match what is already conducted within the company, to not have to adapt too many things. However, sprint length is one of the core elements of Scrum, and should be consistent to ensure the benefits of Scrum are realised.

Recommendation

Process: "Experiment with different Sprint lengths to find out which is a suitable Sprint length in the project. Stick to the length that is agreed between different parties."

ITS: Use the ITS to detect and visualise sprint length over time, to encourage consistency.

Context

Stakeholder Benefits: *Everyone*: Don't need to consider the possibility of varying-length sprints, whether that be for planning, workload, commitment, or other factors.

Stakeholder Costs: *Managers*: Must be strict about not modifying the sprint length.

Artefact Scope: Project/Sprint

Issue Types: All

Inclusion Factors: Team uses Agile (Scrum).

Exclusion Factors: None

Violation

Smells: Sprints have different durations. Sprint does not end at the scheduled time.

Consequences: "As there is no time boxing, visibility of the made progress gets blurrier and the possible problems in the development do not surface. No commitment to sprint goals from the Team as a Sprint just might be extended if it seems that goals will not be met. In addition, there is no exact shipping date for the product increment. This may lead to situation where ready features are laying in the version control system unused for prolonged periods. Moreover, the customer does not know when she will get the promised features. Sprint planning and Sprint review schedules need to be agreed separately causing wasted extra work."

Causes: "Lack of discipline, Customer caused disruption anti-pattern, Customer Product Owner anti-pattern."

Algorithmic Detection: 1) Calculate the length of the sprints from the sprints defined within the ITS. 2) Report those to the appropriate stakeholder or visualise them in a dashboard for stakeholders to review.

Extracted and extended from Eloranta et al. [53, 54] and Telemaco et al. [221, 222]

Table A.40.: BP39: Use Acceptance Criteria.

Summary

Objective: Improve user stories by using acceptance criteria as a measure of completion.

Motivation: "In agile software development, acceptance criteria play a crucial role in defining and verifying the completion of user stories. Acceptance criteria serve as a way of documenting requirements and ensuring that each User Story meets a set of predefined conditions before it can be considered done. It is essential that every issue has at least one acceptance criteria, as this enables the User Story to reach its definition of done and helps to ensure that it meets the expected standards and requirements."

Recommendation

Process: Use acceptance criteria and have at least one on every issue.

ITS: Have a dedicated "acceptance criteria" field on user stories, and make that a mandatory field (on creation or before completion).

Context

Stakeholder Benefits: *Developers*: Confidence in their knowledge of when a User Story is done. *Requirements Engineers or Product Owners*: Confidence that their user stories will be implemented to their satisfaction. *Customers*: An understanding of the actual desired features, since they will know when they have received the feature they requested.

Stakeholder Costs: *Issue Creator*: Must conceptualise and write acceptance criteria. *Developer or Tester*: Must test and verify that acceptance criteria have been met.

Artefact Scope: Issue

Issue Types: User Story

Inclusion Factors: Team uses Agile.

Exclusion Factors: None

Violation

Smells: User stories have no acceptance criteria.

Consequences: Uncertainty and disagreement about the completion of user stories.

Causes: People are unaware of the benefits of acceptance criteria. It takes extra effort to conceptualise and write acceptance criteria.

Algorithmic Detection: Detect missing acceptance criteria and alert the responsible stakeholders (assignee or manager).

Extracted and extended from Prediger [184]

Table A.41.: BP40: Limit Acceptance Criteria.

Summary

Objective: Improve user stories by limiting the number of acceptance criteria.

Motivation: "While having acceptance criteria is crucial in agile software development, it is equally important to avoid having too many."

Recommendation

Process: Limit the number of acceptance criteria. There is no exact number, as it needs to be considered on a per-project basis. Consider the value being brought be each additional acceptance criteria, and stop once you reach diminishing returns.

ITS: Warn the user when they have entered more than X acceptance criteria (X to be determined by the team/organisation), and ask them to reconsider the need for so many.

Context

Stakeholder Benefits: *Developers*: Confidence in their knowledge of when a User Story is done, without difficulties adhering to too many criteria. *Requirements Engineers or Product Owners*: Confidence that their user stories will be implemented to their satisfaction, since they haven't overspecified.

Stakeholder Costs: *Issue Creator*: Must limit how many acceptance criteria they write.

Artefact Scope: Issue

Issue Types: User Story

Inclusion Factors: Team uses Agile.

Exclusion Factors: None

Violation

Smells: User Stories have an excessive number of acceptance criteria.

Consequences: "Overloading a User Story with too many acceptance criteria can make it difficult for the development team to understand and implement it. It can also slow down the development process, as the team must meet every single criteria before considering the User Story complete."

Causes: Desire to over-specify one vision of the implementation. Lack of awareness of the issues caused by having too many.

Algorithmic Detection: You can detect issues with more than X accetance criteria and notify the appropriate stakeholders, but since this number is flexible and context-sensitive it is best to warn users while they are entering the criteria instead.

Extracted and extended from Prediger [184]

Appendix B.

Additional Figures

Many of the Best Practice figures presented in Chapter 8 are showing data limited to just Bug Reports. Here, I present those same figures, but applied across all issue type themes. They are presented here to provide some comparative value to see how similar the results are across the issue type themes, when compared to Bug Reports.

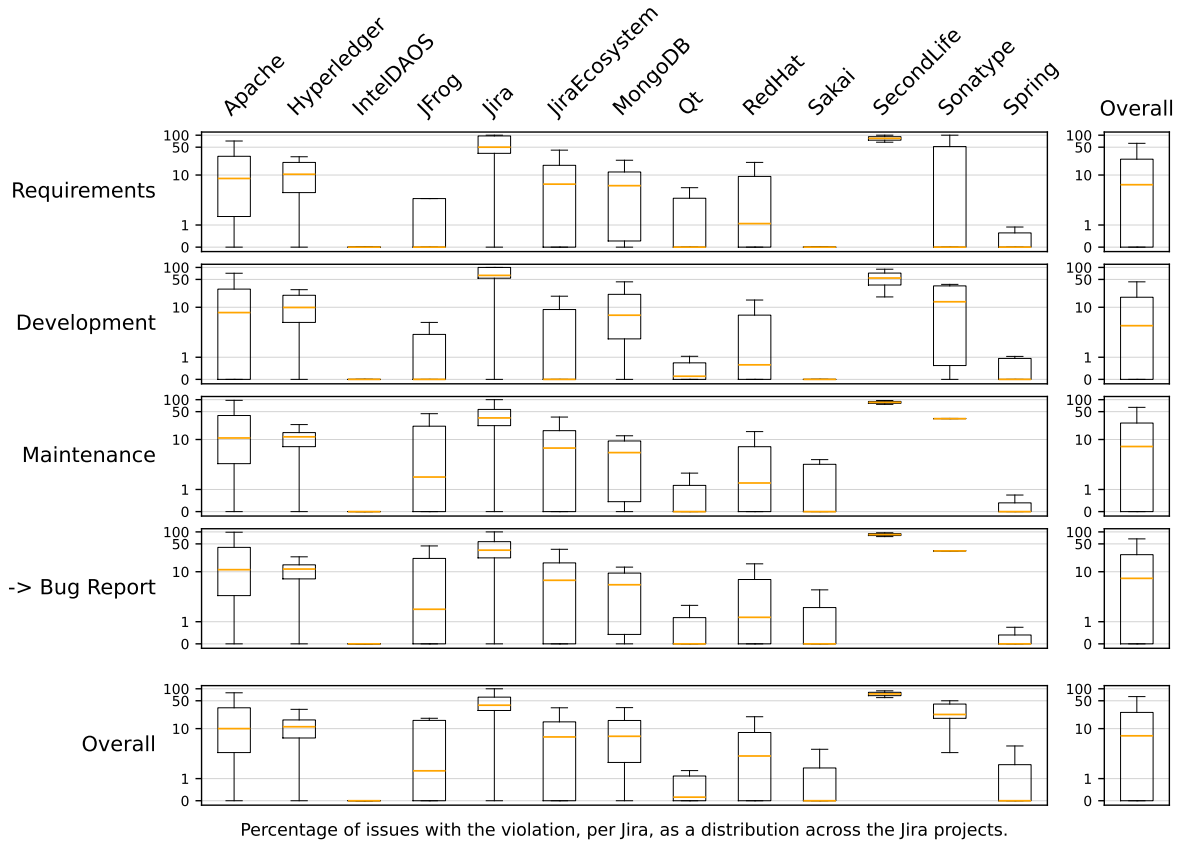


Figure B.1.: Violations to Set Bug Report Assignee (All Themes).

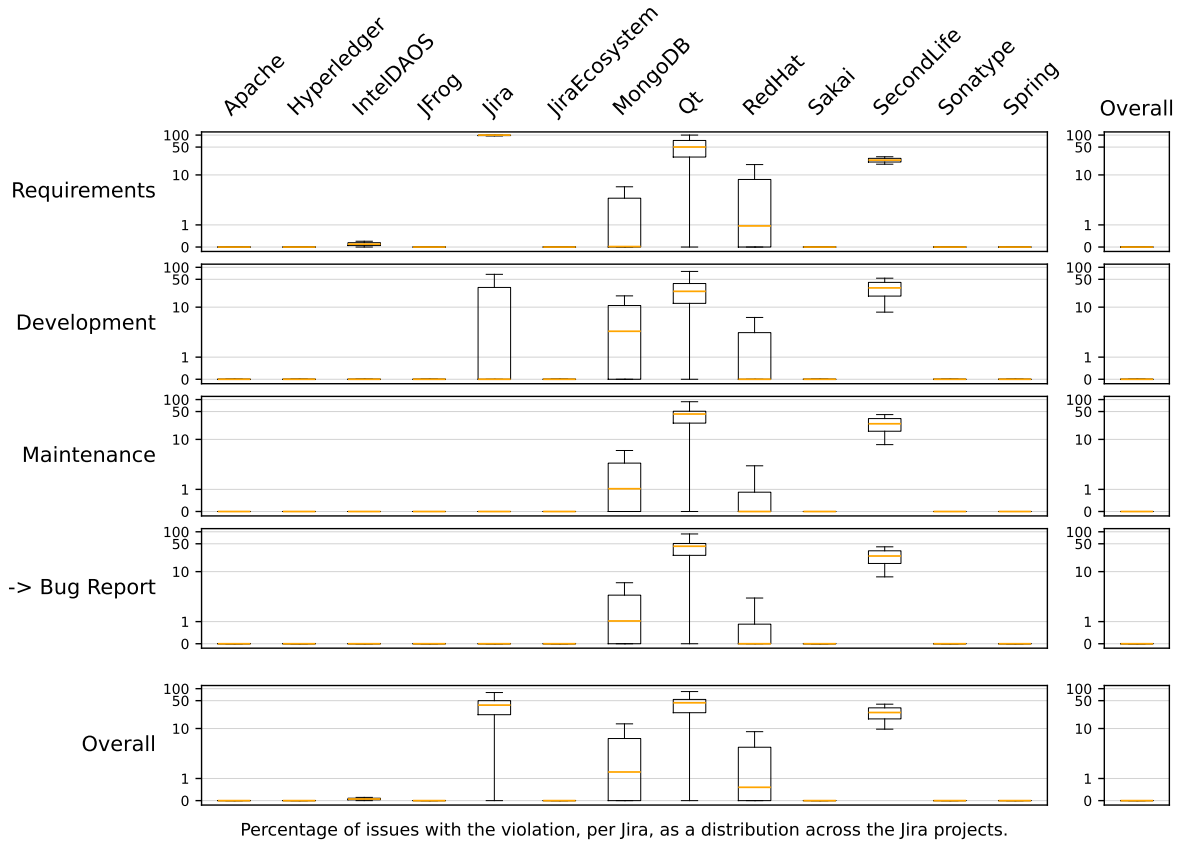


Figure B.2.: Violations to Set Bug Report Priority (All Themes).

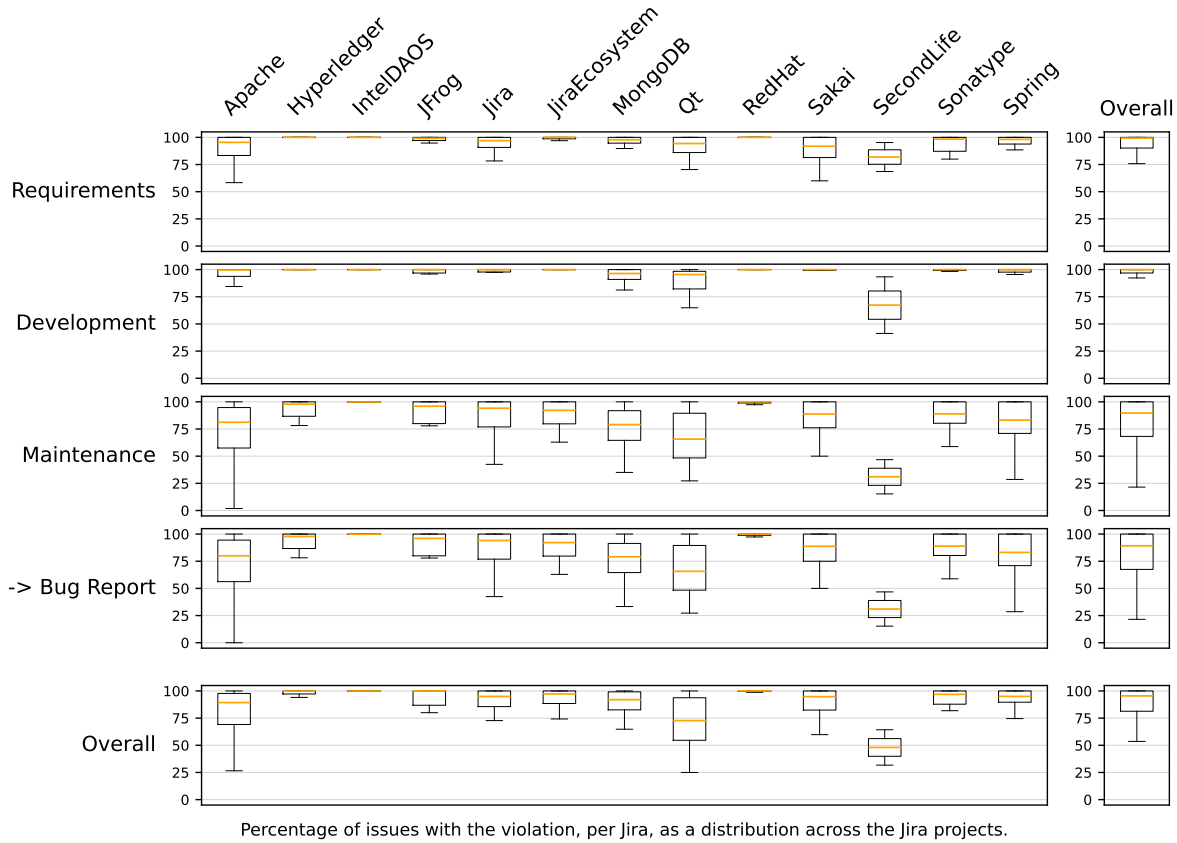


Figure B.3.: Violations to Set Bug Report Environment (All Themes).

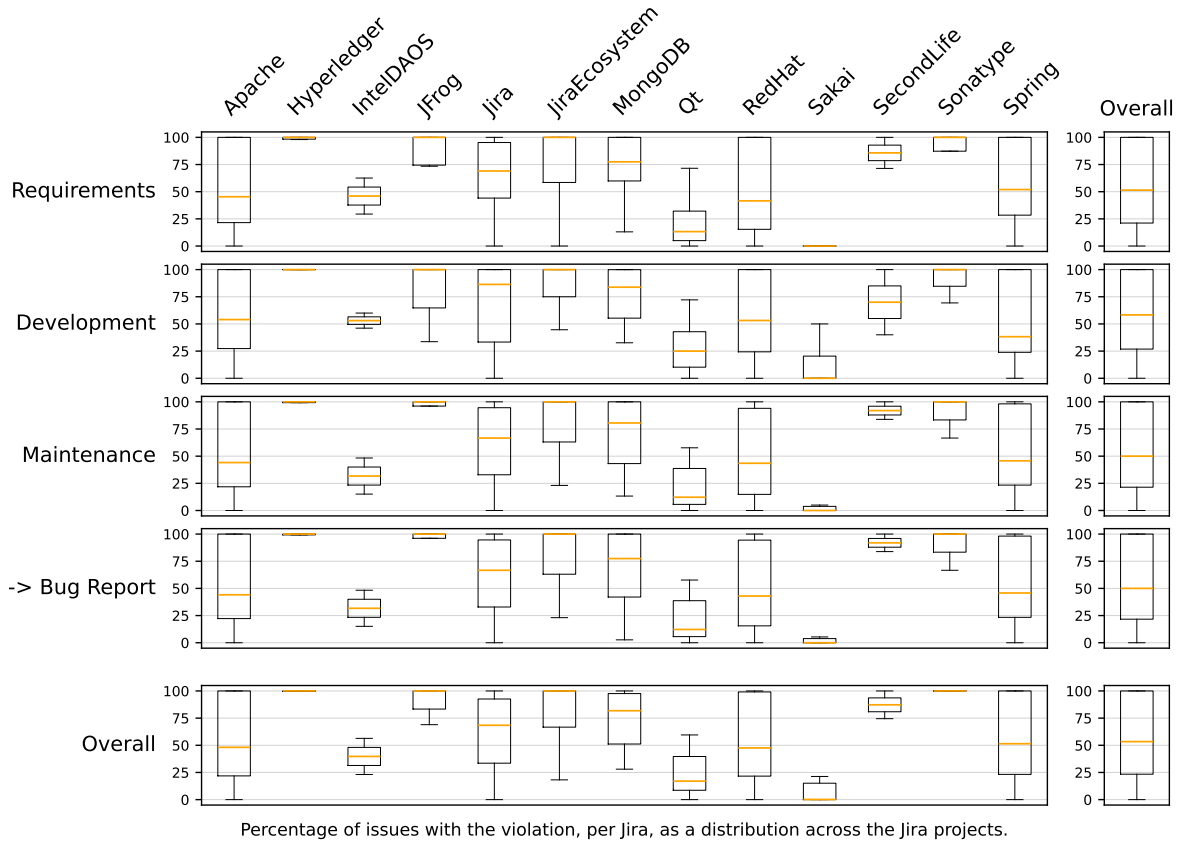


Figure B.4.: Violations to Set Bug Report Components (All Themes).

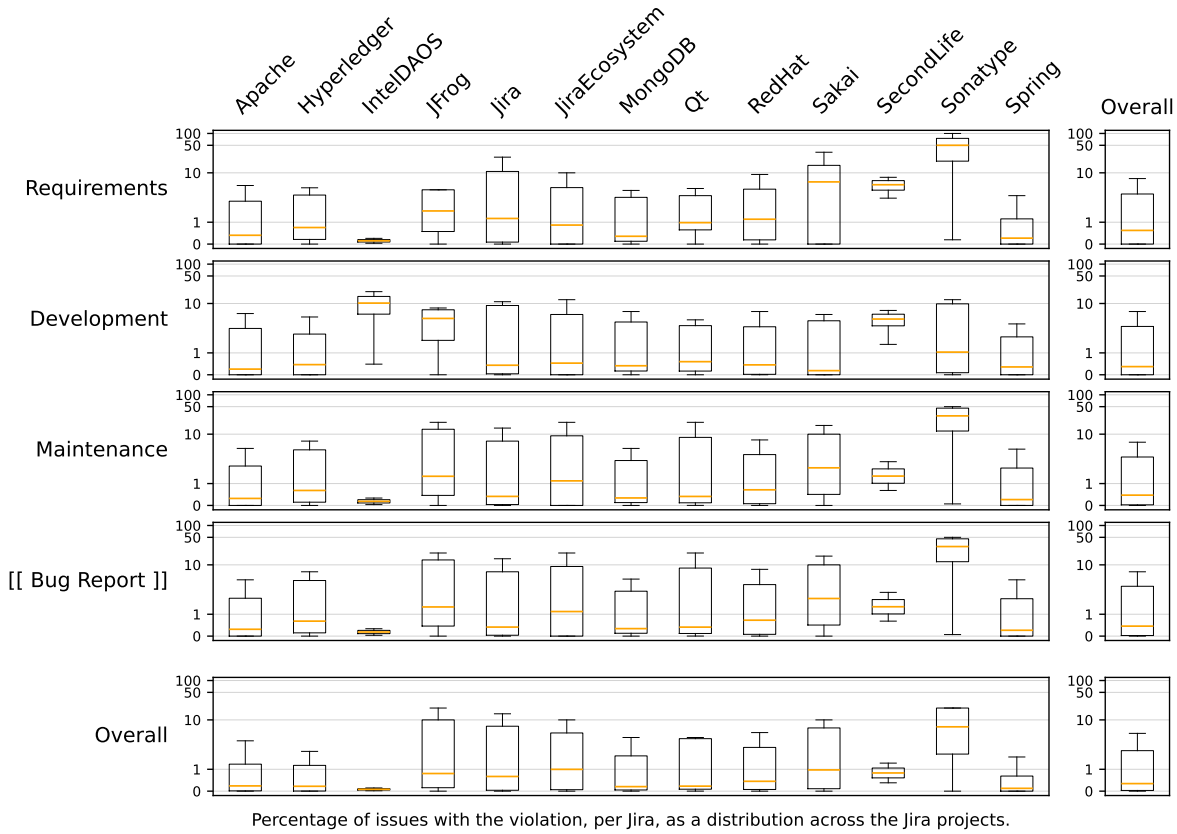


Figure B.5.: Violations to Good First Assignee (All Themes).

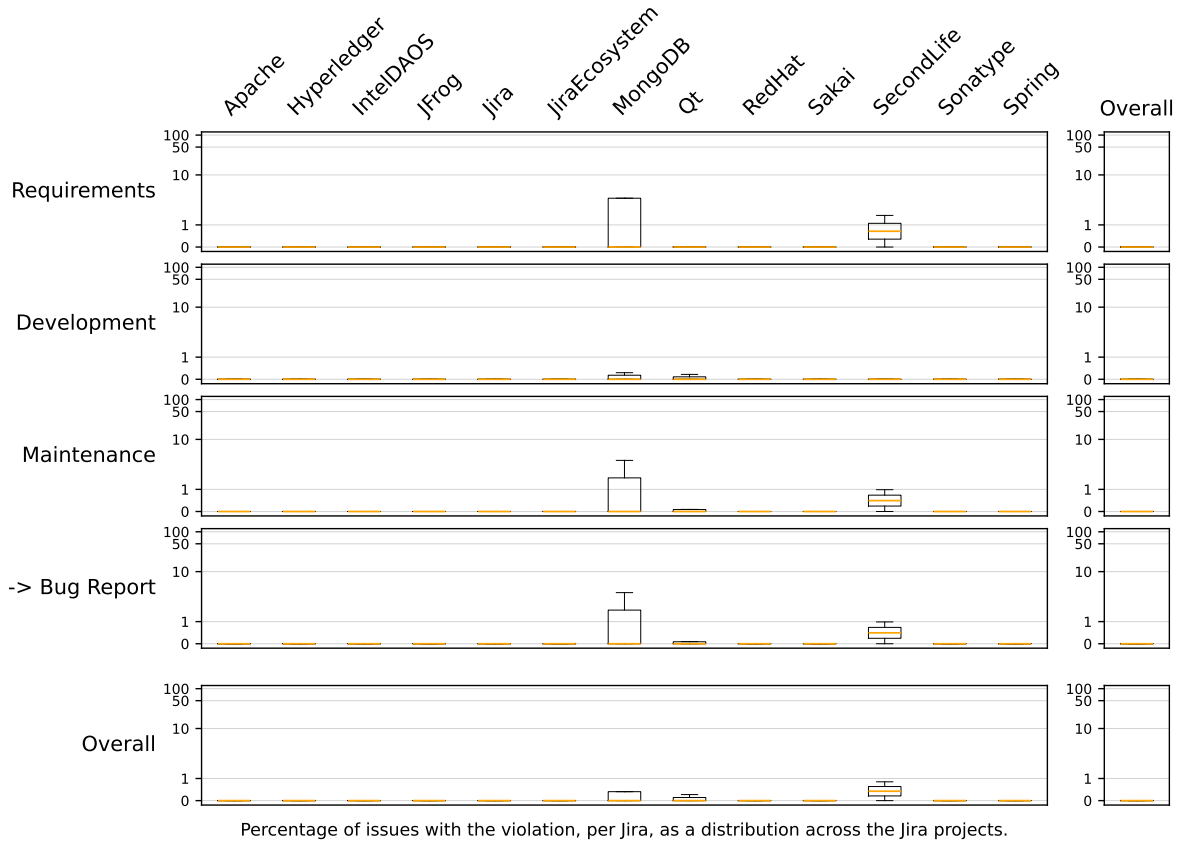


Figure B.6.: Violations to Assign Bugs to Individuals (All Themes).

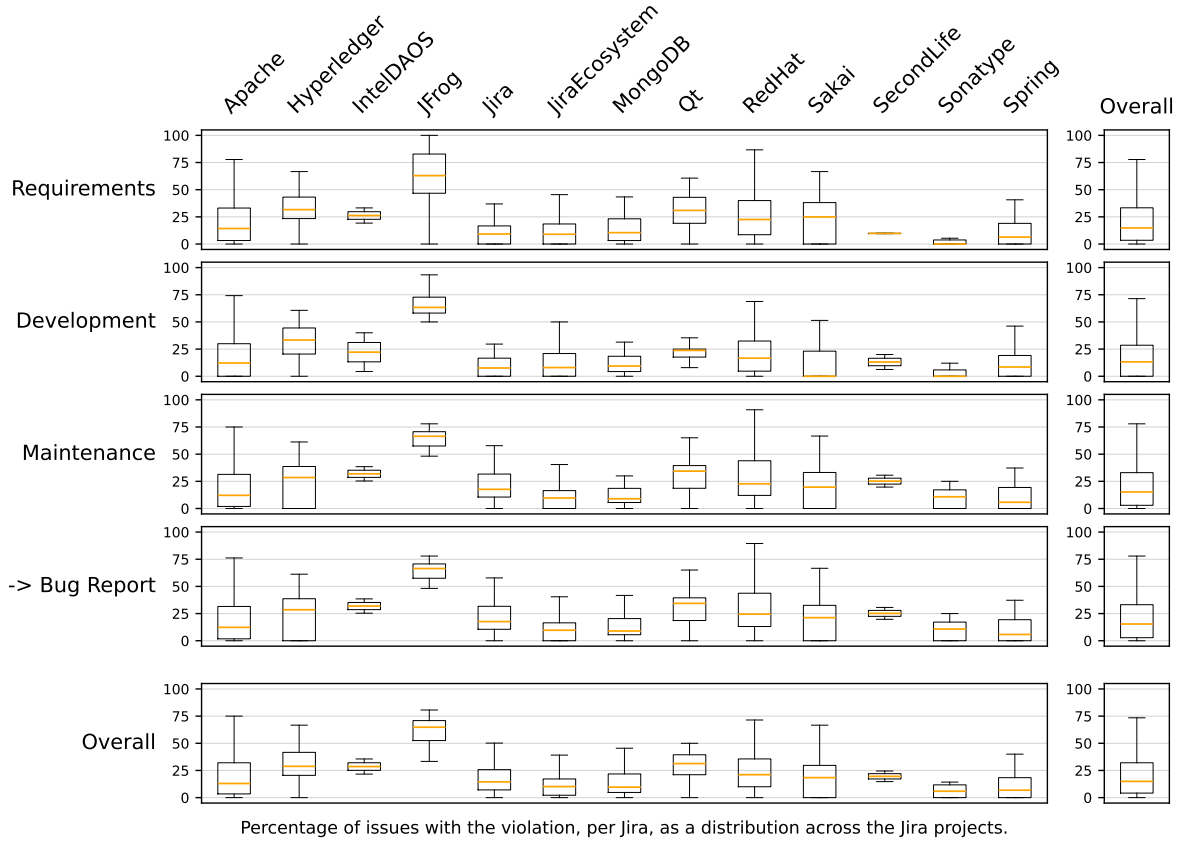


Figure B.7.: Violations to Assignee Bug Resolution (All Themes).

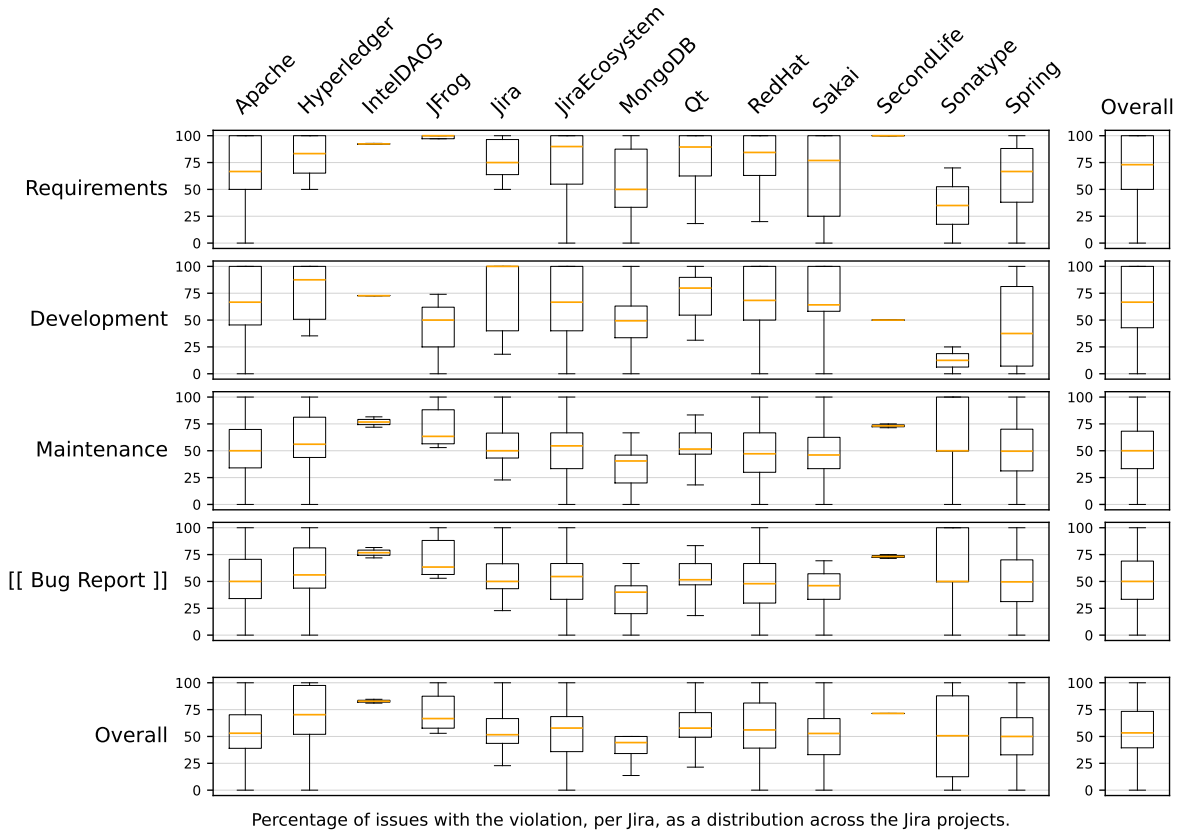


Figure B.8.: Violations to Timely Severe Issue Resolution (All Themes).

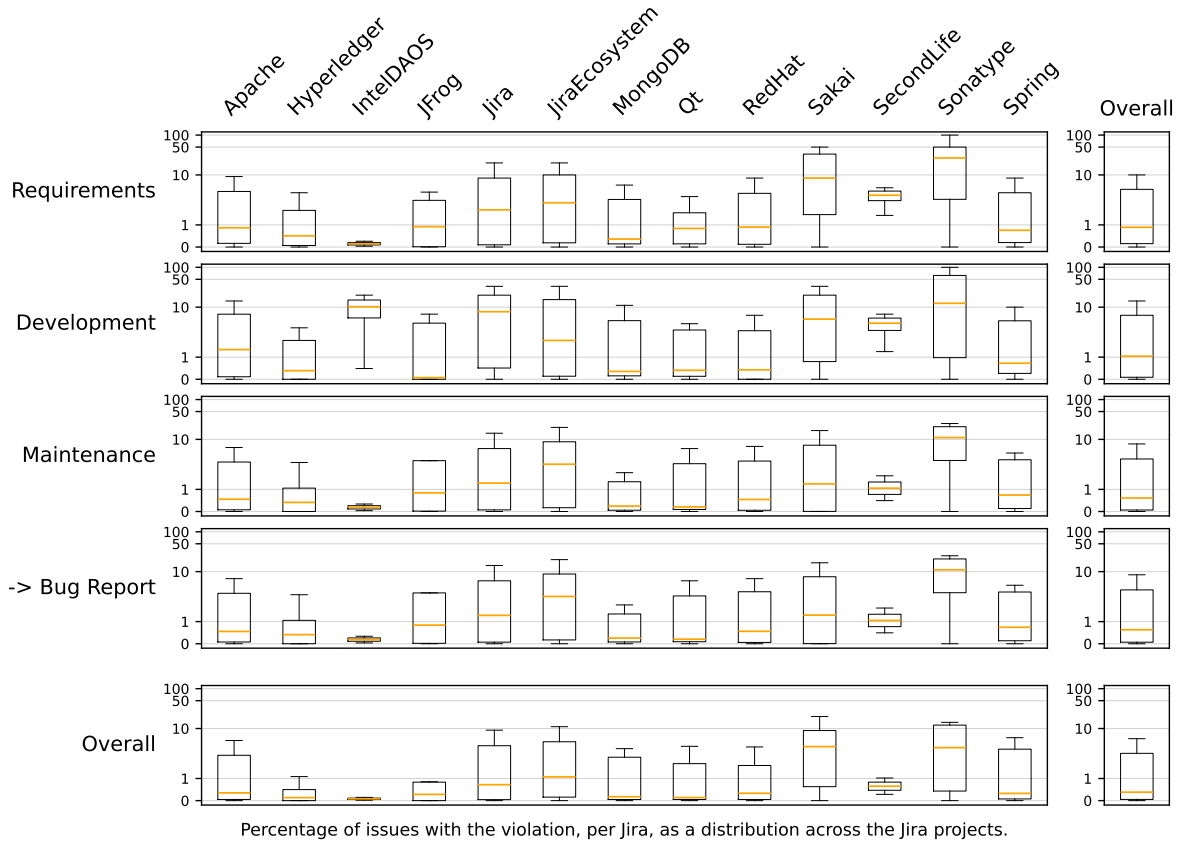


Figure B.9.: Violations to Avoid Zombie Bugs (All Themes).

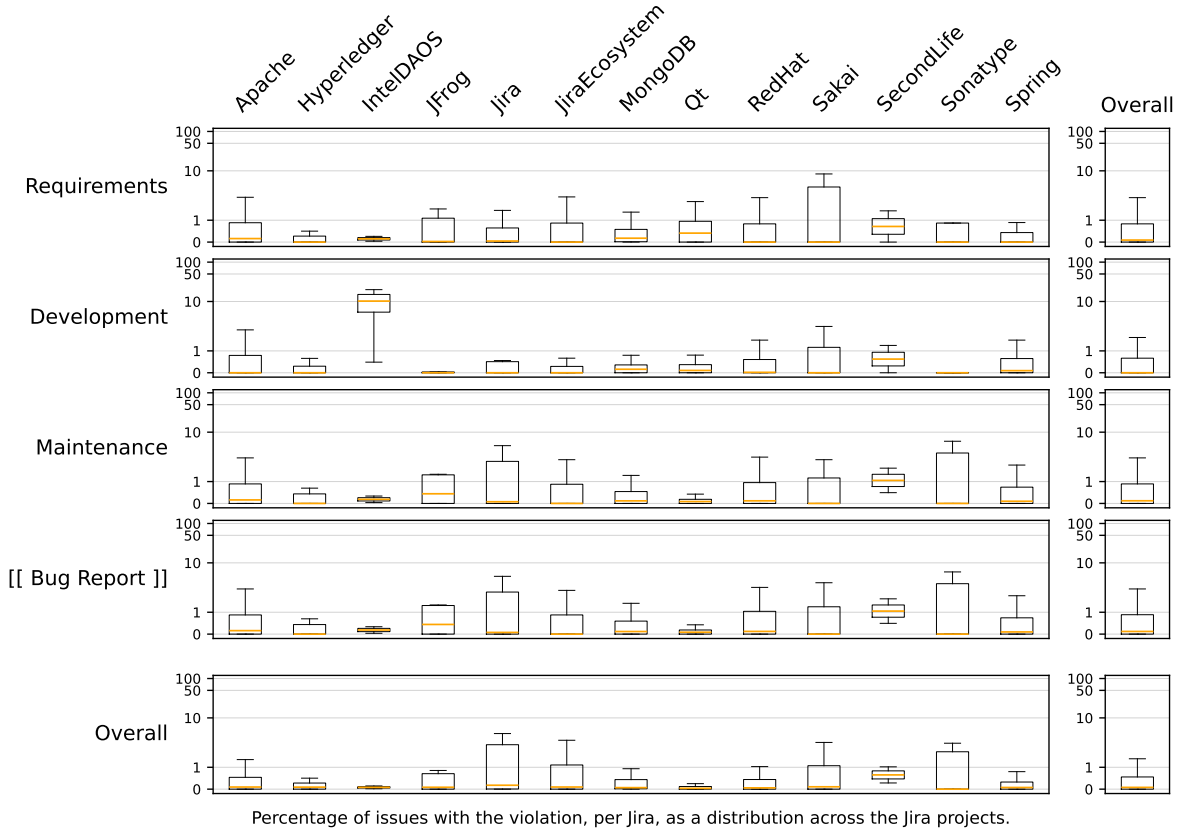


Figure B.10.: Violations to Stable Closed State (All Themes).

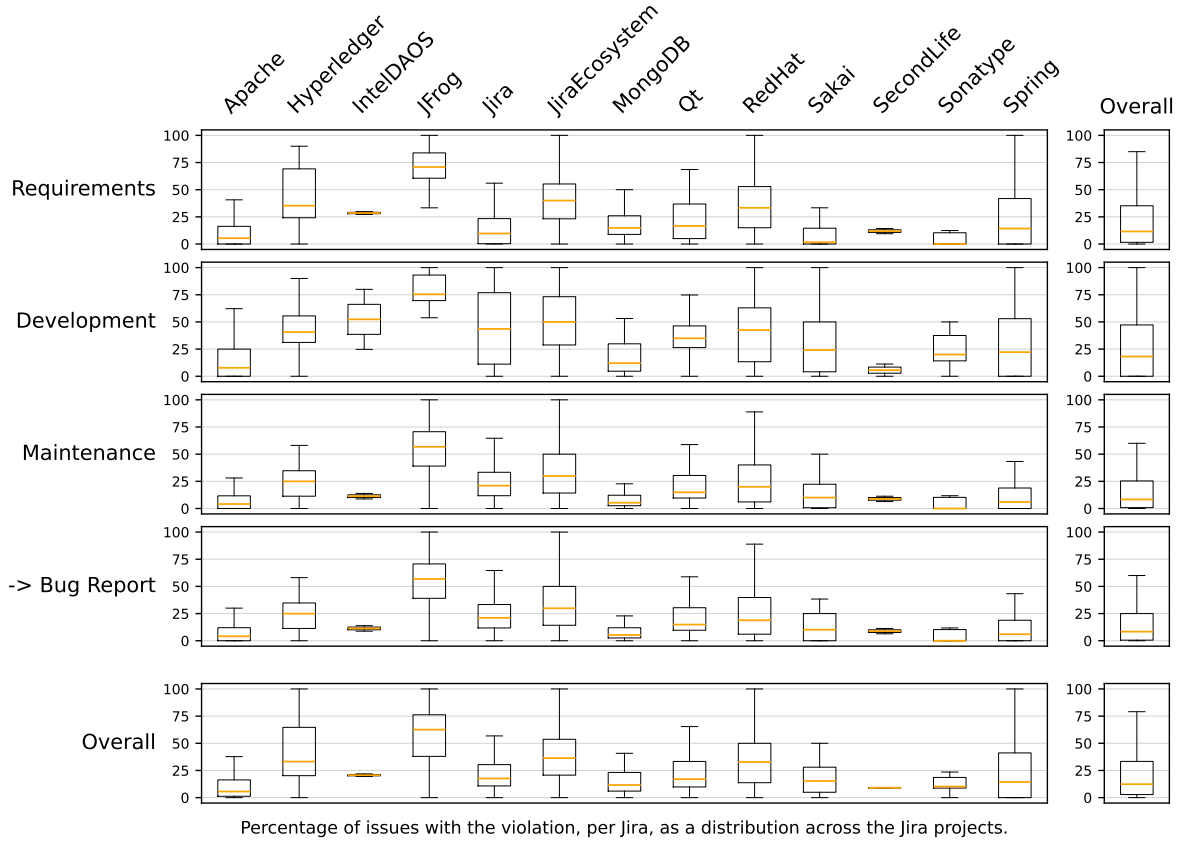


Figure B.11.: Violations to Bug Report Discussion (All Themes).

List of Figures

List of Tables

List of Publications

My publications, central to the thesis, included in this work (partly verbatim):

- Montgomery, L. and Maalej, W. 'Beyond Bug Reports and Feature Requests: How Do Software Projects Manage Different Issue Types?' In: To be submitted... (2025)
- Montgomery, L., Lüders, C., Rahe, C. and Maalej, W. 'Smells? It depends! An Interview Study of Issue Tracking Problems in Practice'. In: To be submitted... (2025)
- Montgomery, L., Lüders, C. and Maalej, W. 'Mining Issue Trackers: Concepts and Techniques'. In: *Handbook of Natural Language Processing for Requirements Engineering*. Ed. by A. Ferrari and G. Deshpande. Cham, Switzerland: Springer Nature Switzerland AG, 2024. Chap. 11, ???–??? [165]
- Montgomery, L., Lüders, C. and Maalej, W. 'An Alternative Issue Tracking Dataset of Public Jira Repositories'. In: *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 73–77. DOI: 10.1145/3524842.3528486 [164]
- Montgomery, L., Fucci, D., Bouraffa, A., Scholz, L. and Maalej, W. 'Empirical research on requirements quality: a systematic mapping study'. In: *Requir. Eng.* 27.2 (2022), pp. 183–209. DOI: 10.1007/S00766-021-00367-Z [163]

My publications, not central to the thesis, produced during my PhD:

- Frattini, J., Montgomery, L., Fucci, D., Unterkalmsteiner, M., Mendez, D. and Fischbach, J. 'Requirements quality research artifacts: Recovery, analysis, and management guideline'. In: *J. Syst. Softw.* 216 (2024), p. 112120. DOI: 10.1016/J.JSS.2024.112120 [78]
- Frattini, J., Fucci, D., Torkar, R., Montgomery, L., Unterkalmsteiner, M., Fischbach, J. and Mendez, D. 'Applying Bayesian Data Analysis for Causal Inference about Requirements Quality: A Replicated Experiment'. In: *CoRR* abs/2401.01154 (2024). DOI: 10.48550/ARXIV.2401.01154. arXiv: 2401.01154 [74]
- Frattini, J., Montgomery, L., Fischbach, J., Mendez, D., Fucci, D. and Unterkalmsteiner, M. 'Requirements quality research: a harmonized theory, evaluation, and roadmap'. In: *Requir. Eng.* 28.4 (2023), pp. 507–520. DOI: 10.1007/S00766-023-00405-Y [75]
- Frattini, J., Montgomery, L., Fucci, D., Fischbach, J., Unterkalmsteiner, M. and Mendez, D. 'Let's Stop Building at the Feet of Giants: Recovering unavailable Requirements Quality Artifacts'. In: *Joint Proceedings of REFSQ-2023 Workshops, Doctoral Symposium, Posters & Tools Track and Journal Early Feedback co-located with the 28th International Conference on Requirements*

Engineering: Foundation for Software Quality (REFSQ 2023), Barcelona, Catalunya, Spain, April 17-20, 2023. Ed. by A. Ferrari. Vol. 3378. CEUR Workshop Proceedings. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3378/NLP4RE-paper3.pdf [77]

- Frattini, J., Montgomery, L., Fischbach, J., Unterkalmsteiner, M., Mendez, D. and Fucci, D. 'A Live Extensible Ontology of Quality Factors for Textual Requirements'. In: *30th IEEE International Requirements Engineering Conference, RE 2022, Melbourne, Australia, August 15-19, 2022*. IEEE, 2022, pp. 274–280. DOI: 10.1109/RE54965.2022.00041 [76]
- Puhlfürß, T., Montgomery, L. and Maalej, W. 'An Exploratory Study of Documentation Strategies for Product Features in Popular GitHub Projects'. In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*. IEEE, 2022, pp. 379–383. DOI: 10.1109/ICSME55016.2022.00043 [186]
- Pham, Y. D., Montgomery, L. and Maalej, W. 'Renovating Requirements Engineering: First Thoughts to Shape Requirements Engineering as a Profession'. In: *27th IEEE International Requirements Engineering Conference Workshops, RE 2019 Workshops, Jeju Island, Korea (South), September 23-27, 2019*. IEEE, 2019, pp. 7–11. DOI: 10.1109/REW.2019.00008 [182]
- Stanik, C., Montgomery, L., Martens, D., Fucci, D. and Maalej, W. 'A Simple NLP-Based Approach to Support Onboarding and Retention in Open Source Communities'. In: *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 172–182. DOI: 10.1109/ICSME.2018.00027 [215]
- Fucci, D., Stanik, C., Montgomery, L., Kurtanovic, Z., Johann, T. and Maalej, W. 'Research on NLP for RE at the University of Hamburg: A Report'. In: *Joint Proceedings of REFSQ-2018 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018), Utrecht, The Netherlands, March 19, 2018*. Ed. by K. Schmid. Vol. 2075. CEUR Workshop Proceedings. CEUR-WS.org, 2018. URL: https://ceur-ws.org/Vol-2075/NLP4RE%5C_paper8.pdf [80]

Abbreviations

ERD	Entity Relationship Diagram
IQR	Interquartile Range
ITE	Issue Tracking Ecosystem
ITS	Issue Tracking System
Jira Repo	Jira Repository
NLP	Natural Language Processing
OSS	Open-Source Software
QA	Quality Assurance
RE	Requirements Engineering
SDD	Software Design Document
SDM	Software Development Methodology
SE	Software Engineering
SMS	Systematic Mapping Study
SPM	Software Process Model
SRS	Software Requirements Specification
UML	Unified Modelling Language

Glossary

Agile Requirements Engineering The approach to Requirements Engineering whereby iteration is a key principle. Instead of gated phases (as in Traditional Requirements Engineering), requirements are gathered and implemented in short cycles of 2–6 weeks. These cycles allow for quick prototyping, customer interaction, and small increments of delivered software.

Change-Based Requirements Engineering The approach to Requirements Engineering whereby the focus is on product maintenance, instead of product creation. The requirements are gathered through various bottom-up requests like Bug Reports and Feature Requests, and very little (if any) roadmap or corporate direction. Instead, it is the users and small group of maintainers who keep the software updated and relevant.

Entity Relationship Diagram A type of UML diagram that illustrates how entities, such as people, objects, or concepts, related to each other within a defined scope.

Information Systems The field within computer science concerned with (and the process of) collecting, processing, storing, and distributing information, usually within large software systems.

Interquartile Range A measure of statistical distribution across a set of data. It is defined as the difference between the 75th and 25th percentiles of the data. In other words, the central 50% of the data.

issue An artefact in an Issue Tracking System (ITS), the represents and individual unit of work to be completed. Each issue has a type, which dictates its role within the ITS. Examples of issue types include User Story, Feature Request, and Bug Report.

Issue Tracking Ecosystem The context that surrounds an Issue Tracking System, including the organisation/team/people who use it, the processes defined to interact with it, and the communication channels that interface with it.

Issue Tracking System Software that manages and maintains lists of issues. Traditionally used by Software companies to build software, but can also be found within other organisational contexts.

Jira Repository Jira is an Issue Tracking System designed by the company Atlassian. A Jira

Repository is a singular instance of Jira, which was created by an organisation and contains their data.

Natural Language Processing The application of computational techniques to the analysis and synthesis of natural language.

Open-Source Software Software that is released under a licence in which the copyright holder grants users the rights to use, change, and distribute the software and its source code.

Requirements Engineering The field within Software Engineering concerned with (and the process of) defining, documenting, and maintaining requirements.

Software Design Document A representation of a software design that is to be used for recording design information, addressing various design concerns, and communicating that information to the design's stakeholders.

Software Development Methodology A process for organising, planning, and controlling the process of developing software systems.

Software Engineering The field within computer science concerned with (and the process of) designing, developing, testing, and maintaining software.

Software Process Model An abstraction of the actual processes being conducted when building software.

Software Requirements Specification A description of a software system to be developed. Often includes functional and non-functional requirements, use cases, important stakeholders, existing environment, and scope.

Thematic Analysis A qualitative data analysis technique for analysing large quantities of textual data, with the goal of producing a taxonomy of information summarising the textual data. The output is a thematic map, which is a set of themes representing the information in the textual data. Each theme in a thematic map also normally has codes that further break down the themes into meaningful insights from the data.

Traditional Requirements Engineering The approach of Requirements Engineering whereby each phase is gated, requiring the previous phase to be complete before the next phase is started. This approach is also called the Waterfall Model, since you cannot go back up the waterfall. In this approach, the entire Requirements Engineering phase is completed before moving on to implementation. In other words, the Requirements Engineering is conducted only and completely up-front.

Bibliography

[1] *6Sense Jira Market Share*. URL: https://6sense.com/tech/productivity/jira-software-market-share (visited on 03/07/2024).

[2] *A website dedicated entirely to quotes and opinions about how they dislike Jira*. URL: https://ifuckinghatejira.com/ (visited on 20/11/2024).

[3] Adhya, E. 'Key elements of an effective style guide in the new age'. In: *Technical Communication* 62.3 (2015), pp. 183–192.

[4] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. and Angel, S. *A Pattern Language - Towns, Buildings, Construction*. Oxford University Press, 1977.

[5] Alsaqqa, S., Sawalha, S. and Abdel-Nabi, H. 'Agile Software Development: Methodologies and Trends'. In: *Int. J. Interact. Mob. Technol.* 14.11 (2020), pp. 246–270. DOI: 10.3991/IJIM.V14I11.13269.

[6] Amoako-Gyampah, K. and White, K. B. 'User involvement and user satisfaction: An exploratory contingency model'. In: *Information & Management* 25.1 (1993), pp. 1–10. DOI: 10.1016/0378-7206(93)90021-K.

[7] Antoniol, G., Ayari, K., Di Penta, M., Khomh, F. and Guéhéneuc, Y. 'Is it a bug or an enhancement?: a text-based approach to classify change requests'. In: *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*. Ed. by M. Chechik, M. R. Vigder and D. A. Stewart. IBM, 2008, p. 23. DOI: 10.1145/1463788.1463819.

[8] Anvik, J., Hiew, L. and Murphy, G. C. 'Coping with an open bug repository'. In: *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005*. Ed. by M. D. Storey, M. G. Burke, L. Cheng and A. van der Hoek. ACM, 2005, pp. 35–39. DOI: 10.1145/1117696.1117704.

[9] Anvik, J., Hiew, L. and Murphy, G. C. 'Who should fix this bug?' In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. Ed. by L. J. Osterweil, H. D. Rombach and M. L. Soffa. ACM, 2006, pp. 361–370. DOI: 10.1145/1134285.1134336.

[10] Aranda, J. and Venolia, G. 'The secret life of bugs: Going past the errors and omissions in software repositories'. In: *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009, pp. 298–308. DOI: 10.1109/ICSE.2009.5070530.

[11] Arora, C., Sabetzadeh, M. and Briand, L. C. 'An empirical study on the potential usefulness of domain models for completeness checking of requirements'. In: *Empir. Softw. Eng.* 24.4 (2019), pp. 2509–2539. DOI: 10.1007/S10664-019-09693-X.

[12] Arya, D. M., Wang, W., Guo, J. L. C. and Cheng, J. 'Analysis and detection of information types of open source software issue discussions'. In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by J. M. Atlee, T. Bultan and J. Whittle. IEEE / ACM, 2019, pp. 454–464. DOI: 10.1109/ICSE.2019.00058.

[13] *asana*. URL: https://asana.com/ (visited on 03/07/2024).

[14] *Azure*. URL: https://azure.microsoft.com/en-us/products/devops/boards (visited on 03/07/2024).

[15] Bachmann, A., Bird, C., Rahman, F., Devanbu, P. T. and Bernstein, A. 'The missing links: bugs and bug-fix commits'. In: *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. Ed. by G. Roman and A. van der Hoek. ACM, 2010, pp. 97–106. DOI: 10.1145/1882291.1882308.

[16] Banerjee, S. and Cukic, B. 'On the Cost of Mining Very Large Open Source Repositories'. In: *1st IEEE/ACM International Workshop on Big Data Software Engineering, BIGDSE 2015, Florence, Italy, May 23, 2015*. Ed. by L. Baresi, T. Menzies, A. Metzger and T. Zimmermann. IEEE Computer Society, 2015, pp. 37–43. DOI: 10.1109/BIGDSE.2015.16.

[17] Bano, M., Zowghi, D. and da Rimini, F. 'User satisfaction and system success: an empirical exploration of user involvement in software development'. In: *Empir. Softw. Eng.* 22.5 (2017), pp. 2339–2372. DOI: 10.1007/S10664-016-9465-1.

[18] *BaseCamp*. URL: https://basecamp.com/ (visited on 03/07/2024).

[19] Bavota, G. and Russo, B. 'A large-scale empirical study on self-admitted technical debt'. In: *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. Ed. by M. Kim, R. Robbes and C. Bird. ACM, 2016, pp. 315–326. DOI: 10.1145/2901739.2901742.

[20] Baysal, O. 'Informing development decisions: from data to information'. In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by D. Notkin, B. H. C. Cheng and K. Pohl. IEEE Computer Society, 2013, pp. 1407–1410. DOI: 10.1109/ICSE.2013.6606729.

[21] Baysal, O., Holmes, R. and Godfrey, M. W. 'Situational awareness: personalizing issue tracking systems'. In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by D. Notkin, B. H. C. Cheng and K. Pohl. IEEE Computer Society, 2013, pp. 1185–1188. DOI: 10.1109/ICSE.2013.6606674.

[22] Beck, K. L., Coplien, J., Crocker, R., Dominick, L., Meszaros, G., Paulisch, F. and Vlissides, J. M. 'Industrial Experience with Design Patterns'. In: *18th International Conference on Software Engineering, Berlin, Germany, March 25-29, 1996, Proceedings*. Ed. by H. D. Rombach, T. S. E. Maibaum and M. V. Zelkowitz. IEEE Computer Society, 1996, pp. 103–114. URL: http://portal.acm.org/citation.cfm?id=227726.227747.

[23] Bernhardt, B. and Singer, T. 'The neural basis of empathy'. In: *Annual review of neuroscience* 35 (2012), pp. 1–23. DOI: 10.1146/annurev-neuro-062111-150536.

[24] Berntzen, M. N., Stray, V., Moe, N. B. and Hoda, R. 'Responding to change over time: A longitudinal case study on changes in coordination mechanisms in large-scale agile'. In: *Empir. Softw. Eng.* 28.5 (2023), p. 114. DOI: 10.1007/S10664-023-10349-0.

[25] Berry, D. M. 'Empirical evaluation of tools for hairy requirements engineering tasks'. In: *Empirical Software Engineering* 26.6 (2021), p. 111.

[26] Bertram, D., Voida, A., Greenberg, S. and Walker, R. J. 'Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams'. In: *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, CSCW 2010, Savannah, Georgia, USA, February 6-10, 2010*. Ed. by K. Inkpen, C. Gutwin and J. C. Tang. ACM, 2010, pp. 291–300. DOI: 10.1145/1718918.1718972.

[27] Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R. and Zimmermann, T. 'What Makes a Good Bug Report?' In: *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. Atlanta Georgia, USA: Association for Computing Machinery, 2008, pp. 208–318. DOI: 10.1145/1453101.1453146.

[28] Bissyandé, T. F., Thung, F., Wang, S., Lo, D., Jiang, L. and Réveillère, L. 'Empirical Evaluation of Bug Linking'. In: *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*. Ed. by A. Cleve, F. Ricca and M. Cerioli. IEEE Computer Society, 2013, pp. 89–98. DOI: 10.1109/CSMR.2013.19.

[29] Bohn, P. 'Exploring LLMs as a Tool for Automated Detection and Correction of Issue Tracker Smells'. MSc Thesis. University of Hamburg, 2024.

[30] Borg, M. 'From Bugs to Decision Support - Leveraging Historical Issue Reports in Software Evolution'. PhD thesis. Lund University, Sweden, 2015. URL: http://lup.lub.lu.se/record/5268091.

[31] Borg, M., Pfahl, D. and Runeson, P. 'Analyzing Networks of Issue Reports'. In: *17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013*. Ed. by A. Cleve, F. Ricca and M. Cerioli. IEEE Computer Society, 2013, pp. 79–88. DOI: 10.1109/CSMR.2013.18.

[32] Bortis, G. and Hoek, A. van der. 'Teambugs: a collaborative bug tracking tool'. In: *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2011, Waikiki, Honolulu, HI, USA, May 21, 2011*. Ed. by M. Cataldo, C. R. B. de Souza, Y. Dittrich, R. Hoda and H. Sharp. ACM, 2011, pp. 69–71. DOI: 10.1145/1984642.1984659.

[33] Braun, V. and Clarke, V. 'Using thematic analysis in psychology'. In: *Qualitative research in psychology* 3.2 (2006), pp. 77–101.

[34] Braun, V. and Clarke, V. *Thematic Analysis: A Practical Guide*. California, USA: SAGE Publications, 2021.

[35] Bretting, S. *Intelligenz*. Licence Agreement: This work is licensed to Lloyd Montgomery for use in his dissertation (contract is private). This artwork may not be used outside the context of this dissertation. The creator of the artwork, Sara Bretting, acknowledges that the dissertation is licensed under CC BY 4.0 and therefore her artwork can be distributed under the same terms while imbedded in the dissertation. 2024.

[36] Breu, S., Premraj, R., Sillito, J. and Zimmermann, T. 'Information needs in bug reports: improving cooperation between developers and users'. In: *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work, CSCW 2010, Savannah, Georgia, USA, February 6-10, 2010*. Ed. by K. Inkpen, C. Gutwin and J. C. Tang. ACM, 2010, pp. 301–310. DOI: 10.1145/1718918.1718973.

[37] Brown, W. H., Malveau, R. C., McCormick, H. and Mowbray, T. J. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[38] *Bugzilla*. URL: https://www.bugzilla.org/ (visited on 03/07/2024).

[39] Carver, J. C., Dieste, O., Kraft, N. A., Lo, D. and Zimmermann, T. 'How Practitioners Perceive the Relevance of ESEM Research'. In: *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016*. ACM, 2016, 56:1–56:10. DOI: 10.1145/2961111.2962597.

[40] Catolino, G., Palomba, F., Tamburri, D. and Serebrenik, A. 'Understanding Community Smells Variability: A Statistical Approach'. In: *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Society, ICSE (SEIS) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 2021, pp. 77–86. DOI: 10.1109/ICSE-SEIS52602.2021.00017.

[41] Cavalcanti, Y. C., Mota Silveira Neto, P. A. da, Carmo Machado, I. do, Vale, T., Almeida, E. S. de and Lemos Meira, S. R. de. 'Challenges and opportunities for software change request repositories: a systematic mapping study'. In: *J. Softw. Evol. Process.* 26.7 (2014), pp. 620–653. DOI: 10.1002/SMR.1639.

[42] Chaparro, O., Bernal-Cárdenas, C., Lu, J., Moran, K., Marcus, A., Di Penta, M., Poshyvanyk, D. and Ng, V. 'Assessing the quality of the steps to reproduce in bug reports'. In: *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. Ed. by M. Dumas, D. Pfahl, S. Apel and A. Russo. ACM, 2019, pp. 86–96. DOI: 10.1145/3338906.3338947.

[43] Cruzes, D. S. and Dyba, T. 'Recommended Steps for Thematic Synthesis in Software Engineering'. In: *2011 International Symposium on Empirical Software Engineering and Measurement*. Banff, Alberta, Canada: IEEE Computer Society, Sept. 2011, pp. 275–284. DOI: 10.1109/ESEM.2011.36.

[44] Dallal, J. A. and Abdin, A. 'Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review'. In: *IEEE Trans. Software Eng.* 44.1 (2018), pp. 44–69. DOI: 10.1109/TSE.2017.2658573.

[45] Dalpiaz, F. *Requirements data sets (user stories)*. Tech. rep. 2018. DOI: 10.17632/7zbk8zsd8y.1.

[46] *Datanyze Jira Market Share*. URL: https://www.datanyze.com/market-share/project-management--217/jira-market-share (visited on 03/07/2024).

[47] Davies, S. and Roper, M. 'What's in a bug report?' In: *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*. ACM, 2014, 26:1–26:10. DOI: 10.1145/2652524.2652541.

[48] Davis, A. M., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A. and Theofanos, M. 'Identifying and measuring quality in a software requirements specification'. In: *Proceedings of the First International Software Metrics Symposium, METRICS 1993, May 21-22, 1993, Balimore, Maryland, USA*. IEEE Computer Society, 1993, pp. 141–152. DOI: 10.1109/METRIC.1993.263792.

[49] De Clercq, D., Haq, I. U. and Azeem, M. U. 'Why happy employees help'. In: *Personnel Review* (2019). DOI: 10.1108/PR-02-2018-0052.

[50] Deshmukh, J., Annervaz, K. M., Podder, S., Sengupta, S. and Dubash, N. 'Towards Accurate Duplicate Bug Retrieval Using Deep Learning Techniques'. In: *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 115–124. DOI: 10.1109/ICSME.2017.69.

[51] Eckhardt, J., Vogelsang, A., Femmer, H. and Mager, P. 'Challenging Incompleteness of Performance Requirements by Sentence Patterns'. In: *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*. IEEE Computer Society, 2016, pp. 46–55. DOI: 10.1109/RE.2016.24.

[52] Eckhardt, J., Vogelsang, A. and Mendez, D. 'Are "non-functional" requirements really non-functional?: an investigation of non-functional requirements in practice'. In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. Ed. by L. K. Dillon, W. Visser and L. A. Williams. ACM, 2016, pp. 832–842. DOI: 10.1145/2884781.2884788.

[53] Eloranta, V., Koskimies, K. and Mikkonen, T. 'Exploring ScrumBut - An empirical study of Scrum anti-patterns'. In: *Inf. Softw. Technol.* 74 (2016), pp. 194–203. DOI: 10.1016/J.INFSOF.2015.12.003.

[54] Eloranta, V., Koskimies, K., Mikkonen, T. and Vuorinen, J. 'Scrum Anti-Patterns - An Empirical Study'. In: *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 1*. Ed. by P. Muenchaisri and G. Rothermel. IEEE Computer Society, 2013, pp. 503–510. DOI: 10.1109/APSEC.2013.72.

[55] *Enlyft Jira Market Share*. URL: https://enlyft.com/tech/products/atlassian-jira (visited on 03/07/2024).

[56] Ernst, N. A. and Murphy, G. C. 'Case Studies in Just-in-Time Requirements Analysis'. In: *2012 Second IEEE International Workshop on Empirical Requirements Engineering (EmpiRE)*. Chicago, IL, USA: IEEE, Sept. 2012, pp. 25–32. DOI: 10.1109/EmpiRE.2012.6347678.

[57] *Eurostat: Statistics Explained - Enterprise size*. https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Enterprise_size. Accessed: 2024-03-18.

[58] Feller, J. and Fitzgerald, B. 'A framework analysis of the open source software development paradigm'. In: *Proceedings of the Twenty-First International Conference on Information Systems, ICIS 2000, Brisbane, Australia, December 10-13, 2000*. Ed. by S. Ang, H. Krcmar, W. J. Orlikowski, P. Weill and J. I. DeGross. Association for Information Systems, 2000, pp. 58–69. URL: http://aisel.aisnet.org/icis2000/7.

[59] Femmer, H., Mendez, D., Wagner, S. and Eder, S. 'Rapid quality assurance with Requirements Smells'. In: *J. Syst. Softw.* 123 (2017), pp. 190–213. DOI: 10.1016/J.JSS.2016.02.047.

[60] Femmer, H., Unterkalmsteiner, M. and Gorschek, T. 'Which Requirements Artifact Quality Defects are Automatically Detectable? A Case Study'. In: *IEEE 25th International Requirements Engineering Conference Workshops, RE 2017 Workshops, Lisbon, Portugal, September 4-8, 2017*. IEEE Computer Society, 2017, pp. 400–406. DOI: 10.1109/REW.2017.18.

[61] Femmer, H. and Vogelsang, A. 'Requirements Quality Is Quality in Use'. In: *IEEE Softw.* 36.3 (2019), pp. 83–91. DOI: 10.1109/MS.2018.110161823.

[62] Ferrari, A. and Esuli, A. 'An NLP approach for cross-domain ambiguity detection in requirements engineering'. In: *Autom. Softw. Eng.* 26.3 (2019), pp. 559–598. DOI: 10.1007/S10515-019-00261-7.

[63] Fischbach, J., Femmer, H., Mendez, D., Fucci, D. and Vogelsang, A. 'What Makes Agile Test Artifacts Useful?: An Activity-Based Quality Model from a Practitioners' Perspective'. In: *ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020*. Ed. by M. T. Baldassarre, F. Lanubile, M. Kalinowski and F. Sarro. ACM, 2020, 41:1–41:10. DOI: 10.1145/3382494.3421462.

[64] Fischbach, J., Frattini, J., Mendez, D., Unterkalmsteiner, M., Femmer, H. and Vogelsang, A. 'How Do Practitioners Interpret Conditionals in Requirements?' In: *Product-Focused Software Process Improvement - 22nd International Conference, PROFES 2021, Turin, Italy, November 26, 2021, Proceedings*. Ed. by L. Ardito, A. Jedlitschka, M. Morisio and M. Torchiano. Vol. 13126. Lecture Notes in Computer Science. Springer, 2021, pp. 85–102. DOI: 10.1007/978-3-030-91452-3_6.

[65] Fischbach, J., Frattini, J., Spaans, A., Kummeth, M., Vogelsang, A., Mendez, D. and Unterkalmsteiner, M. 'Automatic Detection of Causality in Requirement Artifacts: The CiRA Approach'. In: *Requirements Engineering: Foundation for Software Quality - 27th International Working Conference, REFSQ 2021, Essen, Germany, April 12-15, 2021, Proceedings*. Ed. by F. Dalpiaz and P. Spoletini. Vol. 12685. Lecture Notes in Computer Science. Springer, 2021, pp. 19–36. DOI: 10.1007/978-3-030-73128-1_2.

[66] Fischbach, J., Hauptmann, B., Konwitschny, L., Spies, D. and Vogelsang, A. 'Towards Causality Extraction from Requirements'. In: *28th IEEE International Requirements Engineering Conference, RE 2020, Zurich, Switzerland, August 31 - September 4, 2020*. Ed. by T. D. Breaux, A. Zisman, S. Fricker and M. Glinz. IEEE, 2020, pp. 388–393. DOI: 10.1109/RE48521.2020.00053.

[67] Fitzgerald, C., Letier, E. and Finkelstein, A. 'Early failure prediction in feature request management systems'. In: *RE 2011, 19th IEEE International Requirements Engineering Conference, Trento, Italy, August 29 2011 - September 2, 2011*. IEEE Computer Society, 2011, pp. 229–238. DOI: 10.1109/RE.2011.6051658.

[68] Forsgren, N., Storey, M. D., Maddila, C. S., Zimmermann, T., Houck, B. and Butler, J. L. 'The SPACE of Developer Productivity: There's more to it than you think'. In: *ACM Queue* 19.1 (2021), pp. 20–48. DOI: 10.1145/3454122.3454124.

[69] Fowler, M. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999. URL: http://martinfowler.com/books/refactoring.html.

[70] Fowler, M., Highsmith, J. et al. 'The agile manifesto'. In: *Software development* 9.8 (2001), pp. 28–35.

[71] Franch, X., Mendez, D., Oriol, M., Vogelsang, A., Heldal, R., Knauss, E., Travassos, G. H., Carver, J. C., Dieste, O. and Zimmermann, T. 'How do Practitioners Perceive the Relevance of Requirements Engineering Research? An Ongoing Study'. In: *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*. Ed. by A. Moreira, J. Araújo, J. Hayes and B. Paech. IEEE Computer Society, 2017, pp. 382–387. DOI: 10.1109/RE.2017.17.

[72] Franch, X., Mendez, D., Vogelsang, A., Heldal, R., Knauss, E., Oriol, M., Travassos, G. H., Carver, J. C. and Zimmermann, T. 'How do practitioners perceive the relevance of requirements engineering research?' In: *IEEE Transactions on Software Engineering* 48.6 (2020), pp. 1947–1964.

[73] Franch, X., Mendez, D., Vogelsang, A., Heldal, R., Knauss, E., Oriol, M., Travassos, G. H., Carver, J. C. and Zimmermann, T. 'How do Practitioners Perceive the Relevance of Requirements Engineering Research?' In: *IEEE Trans. Software Eng.* 48.6 (2022), pp. 1947–1964. DOI: 10.1109/TSE.2020.3042747.

[74] Frattini, J., Fucci, D., Torkar, R., Montgomery, L., Unterkalmsteiner, M., Fischbach, J. and Mendez, D. 'Applying Bayesian Data Analysis for Causal Inference about Requirements Quality: A Replicated Experiment'. In: *CoRR* abs/2401.01154 (2024). DOI: 10.48550/ARXIV.2401.01154. arXiv: 2401.01154.

[75] Frattini, J., Montgomery, L., Fischbach, J., Mendez, D., Fucci, D. and Unterkalmsteiner, M. 'Requirements quality research: a harmonized theory, evaluation, and roadmap'. In: *Requir. Eng.* 28.4 (2023), pp. 507–520. DOI: 10.1007/S00766-023-00405-Y.

[76] Frattini, J., Montgomery, L., Fischbach, J., Unterkalmsteiner, M., Mendez, D. and Fucci, D. 'A Live Extensible Ontology of Quality Factors for Textual Requirements'. In: *30th IEEE International Requirements Engineering Conference, RE 2022, Melbourne, Aus-*

tralia, August 15-19, 2022. IEEE, 2022, pp. 274–280. DOI: 10.1109/RE54965.2022.00041.

[77] Frattini, J., Montgomery, L., Fucci, D., Fischbach, J., Unterkalmsteiner, M. and Mendez, D. 'Let's Stop Building at the Feet of Giants: Recovering unavailable Requirements Quality Artifacts'. In: *Joint Proceedings of REFSQ-2023 Workshops, Doctoral Symposium, Posters & Tools Track and Journal Early Feedback co-located with the 28th International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2023), Barcelona, Catalunya, Spain, April 17-20, 2023*. Ed. by A. Ferrari. Vol. 3378. CEUR Workshop Proceedings. CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3378/NLP4RE-paper3.pdf.

[78] Frattini, J., Montgomery, L., Fucci, D., Unterkalmsteiner, M., Mendez, D. and Fischbach, J. 'Requirements quality research artifacts: Recovery, analysis, and management guideline'. In: *J. Syst. Softw.* 216 (2024), p. 112120. DOI: 10.1016/J.JSS.2024.112120.

[79] Fucci, D., Palomares, C., Franch, X., Costal, D., Raatikainen, M., Stettinger, M., Kurtanovic, Z., Kojo, T., Koenig, L., Falkner, A. A., Schenner, G., Brasca, F., Männistö, T., Felfernig, A. and Maalej, W. 'Needs and challenges for a platform to support large-scale requirements engineering: a multiple-case study'. In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*. Ed. by M. Oivo, D. Mendez and A. Mockus. ACM, 2018, 19:1–19:10. DOI: 10.1145/3239235.3240498.

[80] Fucci, D., Stanik, C., Montgomery, L., Kurtanovic, Z., Johann, T. and Maalej, W. 'Research on NLP for RE at the University of Hamburg: A Report'. In: *Joint Proceedings of REFSQ-2018 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018), Utrecht, The Netherlands, March 19, 2018*. Ed. by K. Schmid. Vol. 2075. CEUR Workshop Proceedings. CEUR-WS.org, 2018. URL: https://ceur-ws.org/Vol-2075/NLP4RE%5C_paper8.pdf.

[81] Garshol, L. M. 'Metadata? Thesauri? Taxonomies? Topic Maps! Making Sense of it all'. In: *J. Inf. Sci.* 30.4 (2004), pp. 378–391. DOI: 10.1177/0165551504045856.

[82] Gemkow, T., Conzelmann, M., Hartig, K. and Vogelsang, A. 'Automatic Glossary Term Extraction from Large-Scale Requirements Specifications'. In: *26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018*. Ed. by G. Ruhe, W. Maalej and D. Amyot. IEEE Computer Society, 2018, pp. 412–417. DOI: 10.1109/RE.2018.00052.

[83] *GitHub*. URL: https://github.com/ (visited on 03/07/2024).

[84] *GitLab*. URL: https://about.gitlab.com/ (visited on 03/07/2024).

[85] Gotel, O., Cleland-Huang, J., Hayes, J. H., Zisman, A., Egyed, A., Grünbacher, P., Dekhtyar, A., Antoniol, G., Maletic, J. I. and Mäder, P. 'Traceability Fundamentals'. In: *Software and Systems Traceability*. Ed. by J. Cleland-Huang, O. Gotel and A. Zisman. Springer, 2012, pp. 3–22. DOI: 10.1007/978-1-4471-2239-5_1.

[86] Gregor, S. 'The Nature of Theory in Information Systems'. In: *MIS Q.* 30.3 (2006), pp. 611–642. URL: http://misq.org/the-nature-of-theory-in-information-systems.html.

[87] Groen, E. C., Seyff, N., Ali, R., Dalpiaz, F., Dörr, J., Guzman, E., Hosseini, M., Marco, J., Oriol, M., Perini, A. and Stade, M. J. C. 'The Crowd in Requirements Engineering: The Landscape and Challenges'. In: *IEEE Softw.* 34.2 (2017), pp. 44–52. DOI: 10.1109/MS.2017.33.

[88] Group, S. 'The chaos report'. In: *The Standish Group* (1994), pp. 1–16.

[89] Gruber, T. 'A translation approach to portable ontology specifications'. In: *Knowledge acquisition* 5.2 (1993), pp. 199–220.

[90] Gruber, T. 'Toward principles for the design of ontologies used for knowledge sharing?' In: *International journal of human-computer studies* 43.5-6 (1995), pp. 907–928.

[91] Gruber, T. 'What is an Ontology?' In: (1995). Accessed: 2024-08-27.

[92] Gruber, T. 'Ontology'. In: (2009). Accessed: 2024-08-27.

[93] Guzman, E. and Maalej, W. 'How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews'. In: *IEEE 22nd International Requirements Engineering Conference, RE 2014, Karlskrona, Sweden, August 25-29, 2014*. Ed. by T. Gorschek and R. R. Lutz. IEEE Computer Society, 2014, pp. 153–162. DOI: 10.1109/RE.2014.6912257.

[94] Halverson, C. A., Ellis, J. B., Danis, C. and Kellogg, W. A. 'Designing task visualizations to support the coordination of work in software development'. In: *Proceedings of the 2006 ACM Conference on Computer Supported Cooperative Work, CSCW 2006, Banff, Alberta, Canada, November 4-8, 2006*. Ed. by P. J. Hinds and D. Martin. ACM, 2006, pp. 39–48. DOI: 10.1145/1180875.1180883.

[95] Happel, H. and Maalej, W. 'Potentials and challenges of recommendation systems for software development'. In: *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering, RSSE 2008, Atlanta, GA, USA, November 9, 2008*. ACM, 2008, pp. 11–15. DOI: 10.1145/1454247.1454251.

[96] Hassan, A. E. 'The road ahead for mining software repositories'. In: *2008 frontiers of software maintenance*. IEEE. 2008, pp. 48–57.

[97] He, J., Xu, L., Yan, M., Xia, X. and Lei, Y. 'Duplicate Bug Report Detection Using Dual-Channel Convolutional Neural Networks'. In: *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 117–127. DOI: 10.1145/3387904.3389263.

[98] Heck, P. and Zaidman, A. 'An analysis of requirements evolution in open source projects: recommendations for issue trackers'. In: *13th International Workshop on Principles of Software Evolution, IWPSE 2013, Proceedings, August 19-20, 2013, Saint Petersburg, Russia*. Ed. by R. Robbes and G. Robles. ACM, 2013, pp. 43–52. DOI: 10.1145/2501543.2501550.

[99] Heck, P. and Zaidman, A. 'Horizontal traceability for just-in-time requirements: the case for open source feature requests'. In: *J. Softw. Evol. Process.* 26.12 (2014), pp. 1280–1296. DOI: 10.1002/SMR.1678.

[100] Heck, P. and Zaidman, A. 'A framework for quality assessment of just-in-time requirements: the case of open source feature requests'. In: *Requir. Eng.* 22.4 (2017), pp. 453–473. DOI: 10.1007/S00766-016-0247-5.

[101] Herraiz, I., Germán, D. M., González-Barahona, J. M. and Robles, G. 'Towards a simplification of the bug report form in eclipse'. In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings*. Ed. by A. E. Hassan, M. Lanza and M. W. Godfrey. ACM, 2008, pp. 145–148. DOI: 10.1145/1370750.1370786.

[102] Herzig, K., Just, S. and Zeller, A. 'It's not a bug, it's a feature: how misclassification impacts bug prediction'. In: *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. Ed. by D. Notkin, B. H. C. Cheng and K. Pohl. IEEE Computer Society, 2013, pp. 392–401. DOI: 10.1109/ICSE.2013.6606585.

[103] Hesse, T., Lerche, V., Seiler, M., Knöß, K. and Paech, B. 'Documented decision-making strategies and decision knowledge in open source projects: An empirical study on Firefox issue reports'. In: *Inf. Softw. Technol.* 79 (2016), pp. 36–51. DOI: 10.1016/J.INFSOF.2016.06.003.

[104] Huo, D., Ding, T., McMillan, C. and Gethers, M. 'An Empirical Study of the Effects of Expert Knowledge on Bug Reports'. In: *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 1–10. DOI: 10.1109/ICSME.2014.22.

[105] Iftikhar, U., Ali, N. B., Börstler, J. and Usman, M. 'A tertiary study on links between source code metrics and external quality attributes'. In: *Inf. Softw. Technol.* 165 (2024), p. 107348. DOI: 10.1016/J.INFSOF.2023.107348.

[106] Imran, M. M., Ciborowska, A. and Damevski, K. 'Automatically Selecting Follow-up Questions for Deficient Bug Reports'. In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 2021, pp. 167–178. DOI: 10.1109/MSR52588.2021.00029.

[107] Izadi, M., Akbari, K. and Heydarnoori, A. 'Predicting the objective and priority of issue reports in software repositories'. In: *Empir. Softw. Eng.* 27.2 (2022), p. 50. DOI: 10.1007/S10664-021-10085-3.

[108] Janák, J. 'Issue tracking systems'. PhD thesis. Masarykova univerzita, Fakulta informatiky, 2009.

[109] Jayatilleke, S. and Lai, R. 'A systematic review of requirements change management'. In: *Inf. Softw. Technol.* 93 (2018), pp. 163–185. DOI: 10.1016/J.INFSOF.2017.09.004.

[110] Jeong, G., Kim, S. and Zimmermann, T. 'Improving bug triage with bug tossing graphs'. In: *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. Ed. by H. van Vliet and V. Issarny. ACM, 2009, pp. 111–120. DOI: 10.1145/1595696.1595715.

[111] *Jira*. URL: https://www.atlassian.com/software/jira (visited on 03/07/2024).

[112] *Jira Automations*. URL: https://www.atlassian.com/software/jira/guides/expand-jira/automation-use-cases (visited on 03/07/2024).

[113] *Jira Joke on Reddit*. URL: https://www.reddit.com/r/ProgrammerHumor/comments/qt28x3/a_different_level_of_hate/ (visited on 20/11/2024).

[114] *Jira Joke on Reddit: "Happy teams start with Jira, ... and then they are sad."* URL: https://www.reddit.com/r/ProgrammerHumor/comments/1ffpe7f/andthentheyaresad/ (visited on 20/11/2024).

[115] *Jira Jokes on the Atlassian Forum*. URL: https://community.atlassian.com/t5/Watercooler-discussions/Friday-Fun-JIRA-memes/td-p/816534 (visited on 20/11/2024).

[116] Johann, T., Stanik, C., Alizadeh, A. and Maalej, W. 'SAFE: A Simple Approach for Feature Extraction from App Descriptions and App Reviews'. In: *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*. Ed. by A. Moreira, J. Araújo, J. Hayes and B. Paech. IEEE Computer Society, 2017, pp. 21–30. DOI: 10.1109/RE.2017.71.

[117] Johnson, J. N. and Dubois, P. F. 'Issue tracking'. In: *Computing in Science & Engineering* 5.6 (2003), pp. 71–77.

[118] Just, S., Premraj, R. and Zimmermann, T. 'Towards the next generation of bug tracking systems'. In: *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2008, Herrsching am Ammersee, Germany, 15-19 September 2008, Proceedings*. IEEE Computer Society, 2008, pp. 82–85. DOI: 10.1109/VLHCC.2008.4639063.

[119] Kabbedijk, J., Brinkkemper, S., Jansen, S. and Veldt, B. van der. 'Customer Involvement in Requirements Management: Lessons from Mass Market Software Development'. In: *RE 2009, 17th IEEE International Requirements Engineering Conference, Atlanta, Georgia, USA, August 31 - September 4, 2009*. IEEE Computer Society, 2009, pp. 281–286. DOI: 10.1109/RE.2009.28.

[120] Kamata, M. I. and Tamai, T. 'How Does Requirements Quality Relate to Project Success or Failure?' In: *15th IEEE International Requirements Engineering Conference, RE 2007, October 15-19th, 2007, New Delhi, India*. IEEE Computer Society, 2007, pp. 69–78. DOI: 10.1109/RE.2007.31.

[121] Kassab, M., DeFranco, J. F. and Neto, V. V. G. 'An Empirical Investigation on the Satisfaction Levels with the Requirements Engineering Practices: Agile vs. Waterfall'. In: *IEEE International Professional Communication Conference, ProComm 2018, Toronto, ON, Canada, July 22-25, 2018*. IEEE, 2018, pp. 118–124. DOI: 10.1109/PROCOMM.2018.00033.

[122] Kitchenham, B. and Brereton, P. 'A systematic review of systematic review process research in software engineering'. In: *Inf. Softw. Technol.* 55.12 (2013), pp. 2049–2075. DOI: 10.1016/J.INFSOF.2013.07.010.

[123] Kitchenham, B. and Charters, S. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Tech. rep. EBSE-2007-01. Keele University, July 2007, p. 65.

[124] Kuchana, P. *Software architecture design patterns in Java*. Auerbach Publications, 2004.

[125] Kundisch, D., Muntermann, J., Oberländer, A. M., Rau, D., Röglinger, M., Schoormann, T. and Szopinski, D. 'An update for taxonomy designers: methodological guidance from information systems research'. In: *Business & Information Systems Engineering* (2021), pp. 1–19.

[126] Lamkanfi, A., Demeyer, S., Giger, E. and Goethals, B. 'Predicting the severity of a reported bug'. In: *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*. Ed. by J. Whitehead and T. Zimmermann. IEEE Computer Society, 2010, pp. 1–10. DOI: 10.1109/MSR.2010.5463284.

[127] Lamkanfi, A., Demeyer, S., Soetens, Q. D. and Verdonck, T. 'Comparing Mining Algorithms for Predicting the Severity of a Reported Bug'. In: *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*. Ed. by T. Mens, Y. Kanellopoulos and A. Winter. IEEE Computer Society, 2011, pp. 249–258. DOI: 10.1109/CSMR.2011.31.

[128] Laplante, P. A. and Kassab, M. *Requirements engineering for software and systems*. Auerbach Publications, 2022.

[129] Lazar, A., Ritchey, S. and Sharif, B. 'Generating duplicate bug datasets'. In: *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*. Ed. by P. T. Devanbu, S. Kim and M. Pinzger. ACM, 2014, pp. 392–395. DOI: 10.1145/2597073.2597128.

[130] Li, F., Horkoff, J., Liu, L., Borgida, A., Guizzardi, G. and Mylopoulos, J. 'Engineering Requirements with Desiree: An Empirical Evaluation'. In: *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings*. Ed. by S. Nurcan, P. Soffer, M. Bajec and J. Eder. Vol. 9694. Lecture Notes in Computer Science. Springer, 2016, pp. 221–238. DOI: 10.1007/978-3-319-39696-5_14.

[131] Li, J., Zhang, H., Zhu, L., Jeffery, D. R., Wang, Q. and Li, M. 'Preliminary results of a systematic review on requirements evolution'. In: *16th International Conference on Evaluation & Assessment in Software Engineering, EASE 2012, Ciudad Real, Spain, May 14-15, 2012. Proceedings*. Ed. by M. T. Baldassarre, M. Genero, E. Mendes and M. Piattini. IET - The Institute of Engineering and Technology / IEEE Xplore, 2012, pp. 12–21. DOI: 10.1049/IC.2012.0002.

[132] Li, L., Ren, Z., Li, X., Zou, W. and Jiang, H. 'How Are Issue Units Linked? Empirical Study on the Linking Behavior in GitHub'. In: *25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, December 4-7, 2018*. IEEE, 2018, pp. 386–395. DOI: 10.1109/APSEC.2018.00053.

[133] Li, Y. and Maalej, W. 'Which Traceability Visualization Is Suitable in This Context? A Comparative Study'. In: *Requirements Engineering: Foundation for Software Quality - 18th International Working Conference, REFSQ 2012, Essen, Germany, March 19-22, 2012. Proceedings*. Ed. by B. Regnell and D. E. Damian. Vol. 7195. Lecture Notes in Computer Science. Springer, 2012, pp. 194–210. DOI: 10.1007/978-3-642-28714-5_17.

[134] Li, Y., Che, X., Huang, Y., Wang, J., Wang, S., Wang, Y. and Wang, Q. 'A Tale of Two Tasks: Automated Issue Priority Prediction with Deep Multi-task Learning'. In: *ESEM '22: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Helsinki, Finland, September 19 - 23, 2022*. Ed. by F. Madeiral, C.

Lassenius, T. Conte and T. Männistö. ACM, 2022, pp. 1–11. DOI: 10.1145/3544902.3546257.

[135] Lo, D., Nagappan, N. and Zimmermann, T. 'How practitioners perceive the relevance of software engineering research'. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. Ed. by E. D. Nitto, M. Harman and P. Heymans. ACM, 2015, pp. 415–425. DOI: 10.1145/2786805.2786809.

[136] Lucassen, G., Dalpiaz, F., Werf, J. M. E. M. van der and Brinkkemper, S. 'The Use and Effectiveness of User Stories in Practice'. In: *Requirements Engineering: Foundation for Software Quality - 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings*. Ed. by M. Daneva and O. Pastor. Vol. 9619. Lecture Notes in Computer Science. Springer, 2016, pp. 205–222. DOI: 10.1007/978-3-319-30282-9_14.

[137] Lüders, C. 'Mining and Understanding Issue Links Towards a Better Issue Management'. PhD Thesis. University of Hamburg, Germany, 2023. URL: https://ediss.sub.uni-hamburg.de/handle/ediss/10310.

[138] Lüders, C., Bouraffa, A. and Maalej, W. 'Beyond Duplicates: Towards Understanding and Predicting Link Types in Issue Tracking Systems'. In: *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 48–60. DOI: 10.1145/3524842.3528457.

[139] Lüders, C., Pietz, T. and Maalej, W. 'Automated Detection of Typed Links in Issue Trackers'. In: *30th IEEE International Requirements Engineering Conference, RE 2022, Melbourne, Australia, August 15-19, 2022*. IEEE, 2022, pp. 26–38. DOI: 10.1109/RE54965.2022.00010.

[140] Lüders, C., Raatikainen, M., Motger, J. and Maalej, W. 'OpenReq Issue Link Map: A Tool to Visualize Issue Links in Jira'. In: *27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019*. Ed. by D. E. Damian, A. Perini and S. Lee. IEEE, 2019, pp. 492–493. DOI: 10.1109/RE.2019.00070.

[141] Maalej, W. 'Task-First or Context-First? Tool Integration Revisited'. In: *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*. IEEE Computer Society, 2009, pp. 344–355. DOI: 10.1109/ASE.2009.36.

[142] Maalej, W., Kurtanovic, Z. and Felfernig, A. 'What stakeholders need to know about requirements'. In: *4th IEEE International Workshop on Empirical Requirements Engineering, EmpiRE 2014, Karlskrona, Sweden, August 25, 2014*. Ed. by M. Daneva,

R. Berntsson-Svensson, X. Franch, N. H. Madhavji and S. Marczak. IEEE Computer Society, 2014, pp. 64–71. DOI: 10.1109/EMPIRE.2014.6890118.

[143] Maalej, W., Kurtanovic, Z., Nabil, H. and Stanik, C. 'On the automatic classification of app reviews'. In: *Requir. Eng.* 21.3 (2016), pp. 311–331. DOI: 10.1007/S00766-016-0251-9.

[144] Maalej, W. and Nabil, H. 'Bug report, feature request, or simply praise? On automatically classifying app reviews'. In: *23rd IEEE International Requirements Engineering Conference, RE 2015, Ottawa, ON, Canada, August 24-28, 2015*. Ed. by D. Zowghi, V. Gervasi and D. Amyot. IEEE Computer Society, 2015, pp. 116–125. DOI: 10.1109/RE.2015.7320414.

[145] Maalej, W., Nayebi, M., Johann, T. and Ruhe, G. 'Toward Data-Driven Requirements Engineering'. In: *IEEE Softw.* 33.1 (2016), pp. 48–54. DOI: 10.1109/MS.2015.153.

[146] W. Maalej and A. K. Thurimella, eds. *Managing Requirements Knowledge*. Springer, 2013. DOI: 10.1007/978-3-642-34419-0.

[147] Madampe, K., Hoda, R. and Grundy, J. C. 'A Faceted Taxonomy of Requirements Changes in Agile Contexts'. In: *IEEE Trans. Software Eng.* 48.10 (2022), pp. 3737–3752. DOI: 10.1109/TSE.2021.3104732.

[148] *Mantis*. URL: https://mantisbt.org/ (visited on 03/07/2024).

[149] Mendez, D. *Requirements Engineering Lecture Series*. 2021. DOI: 10.6084/m9.figshare.17128613.v1.

[150] Mendez, D., Wagner, S., Kalinowski, M., Felderer, M., Mafra, P., Vetrò, A., Conte, T., Christiansson, M., Greer, D., Lassenius, C., Männistö, T., Nayebi, M., Oivo, M., Penzenstadler, B., Pfahl, D., Prikladnicki, R., Ruhe, G., Schekelmann, A., Sen, S., Spínola, R. O., Tuzcu, A., Vara, J. L. de la and Wieringa, R. J. 'Naming the pain in requirements engineering - Contemporary problems, causes, and effects in practice'. In: *Empirical Sofware Engineering* 22.5 (2017), pp. 2298–2338. DOI: 10.1007/S10664-016-9451-7.

[151] Menzies, T. and Marcus, A. 'Automated severity assessment of software defect reports'. In: *24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China*. IEEE Computer Society, 2008, pp. 346–355. DOI: 10.1109/ICSM.2008.4658083.

[152] Merten, T., Falis, M., Hübner, P., Quirchmayr, T., Bürsner, S. and Paech, B. 'Software Feature Request Detection in Issue Tracking Systems'. In: *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*. IEEE Computer Society, 2016, pp. 166–175. DOI: 10.1109/RE.2016.8.

[153] Merten, T., Krämer, D., Mager, B., Schell, P., Bürsner, S. and Paech, B. 'Do Information Retrieval Algorithms for Automated Traceability Perform Effectively on Issue Tracking System Data?' In: *Requirements Engineering: Foundation for Software Quality - 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings*. Ed. by M. Daneva and O. Pastor. Vol. 9619. Lecture Notes in Computer Science. Springer, 2016, pp. 45–62. DOI: 10.1007/978-3-319-30282-9_4.

[154] Meyer, A. N., Barr, E. T., Bird, C. and Zimmermann, T. 'Today Was a Good Day: The Daily Life of Software Developers'. In: *IEEE Trans. Software Eng.* 47.5 (2021), pp. 863–880. DOI: 10.1109/TSE.2019.2904957.

[155] Meyer, A. N., Fritz, T., Murphy, G. C. and Zimmermann, T. 'Software developers' perceptions of productivity'. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by S. Cheung, A. Orso and M. D. Storey. ACM, 2014, pp. 19–29. DOI: 10.1145/2635868.2635892.

[156] *Miro*. URL: https://Miro.com/ (visited on 03/07/2024).

[157] Mockus, A., Fielding, R. T. and Herbsleb, J. D. 'Two case studies of open source software development: Apache and Mozilla'. In: *ACM Trans. Softw. Eng. Methodol.* 11.3 (2002), pp. 309–346. DOI: 10.1145/567793.567795.

[158] *Monday tool*. URL: https://monday.com (visited on 20/11/2024).

[159] Montgomery, L. 'Escalation prediction using feature engineering: addressing support ticket escalations within IBM's ecosystem'. MSc Thesis. University of Victoria, 2017.

[160] Montgomery, L. *Replication Package for PhD Dissertation*. URL: https://doi.org/10.5281/zenodo.14669551 (visited on 17/01/2025).

[161] Montgomery, L. and Damian, D. E. 'What do Support Analysts Know About Their Customers? On the Study and Prediction of Support Ticket Escalations in Large Software Organizations'. In: *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*. Ed. by A. Moreira, J. Araújo, J. Hayes and B. Paech. IEEE Computer Society, 2017, pp. 362–371. DOI: 10.1109/RE.2017.61.

[162] Montgomery, L., Damian, D. E., Bulmer, T. and Quader, S. 'Customer support ticket escalation prediction using feature engineering'. In: *Requir. Eng.* 23.3 (2018), pp. 333–355. DOI: 10.1007/S00766-018-0292-3.

[163] Montgomery, L., Fucci, D., Bouraffa, A., Scholz, L. and Maalej, W. 'Empirical research on requirements quality: a systematic mapping study'. In: *Requir. Eng.* 27.2 (2022), pp. 183–209. DOI: 10.1007/S00766-021-00367-Z.

[164] Montgomery, L., Lüders, C. and Maalej, W. 'An Alternative Issue Tracking Dataset of Public Jira Repositories'. In: *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 73–77. DOI: 10.1145/3524842.3528486.

[165] Montgomery, L., Lüders, C. and Maalej, W. 'Mining Issue Trackers: Concepts and Techniques'. In: *Handbook of Natural Language Processing for Requirements Engineering*. Ed. by A. Ferrari and G. Deshpande. Cham, Switzerland: Springer Nature Switzerland AG, 2024. Chap. 11, ???–???

[166] Montgomery, L., Reading, E. and Damian, D. E. 'ECrits - Visualizing Support Ticket Escalation Risk'. In: *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*. Ed. by A. Moreira, J. Araújo, J. Hayes and B. Paech. IEEE Computer Society, 2017, pp. 452–455. DOI: 10.1109/RE.2017.62.

[167] Nickerson, R. C., Varshney, U. and Muntermann, J. 'A method for taxonomy development and its application in information systems'. In: *Eur. J. Inf. Syst.* 22.3 (2013), pp. 336–359. DOI: 10.1057/EJIS.2012.26.

[168] *Online Jira discussion about missing severity field.* URL: https://community.atlassian.com/t5/Jira-questions/Remind-me-why-Jira-removed-the-Severity-field/qaq-p/773901 (visited on 15/11/2024).

[169] *Online Jira discussion about missing severity field.* URL: https://community.atlassian.com/t5/Jira-questions/Why-there-is-no-Severity-Field-for-Issue-Type-Bug/qaq-p/1117643 (visited on 15/11/2024).

[170] *Online Jira discussion about missing severity field.* URL: https://community.atlassian.com/t5/Jira-questions/where-to-find-jira-issue-severity/qaq-p/1696032 (visited on 15/11/2024).

[171] OpenReq. *Homepage*. URL: https://openreq.eu/ (visited on 03/01/2023).

[172] OpenReq. *Project GitHub Site*. URL: https://github.com/openreqeu (visited on 03/01/2023).

[173] Ortu, M., Destefanis, G., Adams, B., Murgia, A., Marchesi, M. and Tonelli, R. 'The JIRA Repository Dataset: Understanding Social Aspects of Software Development'. In: *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2015, Beijing, China, October 21, 2015*. Ed. by A. Bener, L. L. Minku and B. Turhan. ACM, 2015, 1:1–1:4. DOI: 10.1145/2810146.2810147.

[174] Pagano, D. and Maalej, W. 'User feedback in the appstore: An empirical study'. In: *21st IEEE International Requirements Engineering Conference, RE 2013, Rio de Janeiro-RJ, Brazil, July 15-19, 2013*. IEEE Computer Society, 2013, pp. 125–134. DOI: 10.1109/RE.2013.6636712.

[175] Palomba, F., Tamburri, D., Fontana, F. A., Oliveto, R., Zaidman, A. and Serebrenik, A. 'Beyond Technical Aspects: How Do Community Smells Influence the Intensity of Code Smells?' In: *IEEE Trans. Software Eng.* 47.1 (2021), pp. 108–129. DOI: 10.1109/TSE.2018.2883603.

[176] Parra, E., Alonso, L., Mendieta, R. and Vara, J. L. de la. 'Advances in Artefact Quality Analysis for Safety-Critical Systems'. In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Oct. 2019, pp. 79–84. DOI: 10.1109/ISSREW.2019.00047.

[177] Parra, E., Vara, J. L. de la and Alonso, L. 'Analysis of requirements quality evolution'. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by M. Chaudron, I. Crnkovic, M. Chechik and M. Harman. ACM, 2018, pp. 199–200. DOI: 10.1145/3183440.3195095.

[178] Perez, Q., Jean, P., Urtado, C. and Vauttier, S. 'Bug or not bug? That is the question'. In: *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 2021, pp. 47–58. DOI: 10.1109/ICPC52881.2021.00014.

[179] Petersen, K., Feldt, R., Mujtaba, S. and Mattsson, M. 'Systematic Mapping Studies in Software Engineering'. In: *12th International Conference on Evaluation and Assessment in Software Engineering, EASE 2008, University of Bari, Italy, 26-27 June 2008*. Ed. by G. Visaggio, M. T. Baldassarre, S. G. Linkman and M. Turner. Workshops in Computing. BCS, 2008. URL: http://ewic.bcs.org/content/ConWebDoc/19543.

[180] Petersen, K., Vakkalanka, S. and Kuzniarz, L. 'Guidelines for conducting systematic mapping studies in software engineering: An update'. In: *Inf. Softw. Technol.* 64 (2015), pp. 1–18. DOI: 10.1016/J.INFSOF.2015.03.007.

[181] Petersen, K., Wohlin, C. and Baca, D. 'The Waterfall Model in Large-Scale Development'. In: *Product-Focused Software Process Improvement, 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009. Proceedings*. Ed. by F. Bomarius, M. Oivo, P. Jaring and P. Abrahamsson. Vol. 32. Lecture Notes in Business Information Processing. Springer, 2009, pp. 386–400. DOI: 10.1007/978-3-642-02152-7_29.

[182] Pham, Y. D., Montgomery, L. and Maalej, W. 'Renovating Requirements Engineering: First Thoughts to Shape Requirements Engineering as a Profession'. In: *27th IEEE International Requirements Engineering Conference Workshops, RE 2019 Workshops,*

Jeju Island, Korea (South), September 23-27, 2019. IEEE, 2019, pp. 7–11. DOI: 10.1109/REW.2019.00008.

[183] Pohl, K. *Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam-foundation level-IREB compliant*. Rocky Nook, Inc., 2016.

[184] Prediger, N. 'Visualising Data and Best Practices in Jira Issue Repositories'. MSc Thesis. University of Hamburg, 2023.

[185] Pudlitz, F., Brokhausen, F. and Vogelsang, A. 'Extraction of System States from Natural Language Requirements'. In: *27th IEEE International Requirements Engineering Conference, RE 2019, Jeju Island, Korea (South), September 23-27, 2019*. Ed. by D. E. Damian, A. Perini and S. Lee. IEEE, 2019, pp. 211–222. DOI: 10.1109/RE.2019.00031.

[186] Puhlfürß, T., Montgomery, L. and Maalej, W. 'An Exploratory Study of Documentation Strategies for Product Features in Popular GitHub Projects'. In: *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*. IEEE, 2022, pp. 379–383. DOI: 10.1109/ICSME55016.2022.00043.

[187] Qamar, K. A., Sülün, E. and Tüzün, E. 'Towards a Taxonomy of Bug Tracking Process Smells: A Quantitative Analysis'. In: *47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021, Palermo, Italy, September 1-3, 2021*. Ed. by M. T. Baldassarre, G. Scanniello and A. Skavhaug. IEEE, 2021, pp. 138–147. DOI: 10.1109/SEAA53835.2021.00026.

[188] Qamar, K. A., Sülün, E. and Tüzün, E. 'Taxonomy of bug tracking process smells: Perceptions of practitioners and an empirical analysis'. In: *Inf. Softw. Technol.* 150 (2022), p. 106972. DOI: 10.1016/J.INFSOF.2022.106972.

[189] Ralph, P. 'Toward Methodological Guidelines for Process Theories and Taxonomies in Software Engineering'. In: *IEEE Trans. Software Eng.* 45.7 (2019), pp. 712–735. DOI: 10.1109/TSE.2018.2796554.

[190] Razzaq, A., Buckley, J., Lai, Q., Yu, T. and Botterweck, G. 'A Systematic Literature Review on the Influence of Enhanced Developer Experience on Developers' Productivity: Factors, Practices, and Recommendations'. In: *ACM Comput. Surv.* 57.1 (2025), 13:1–13:46. DOI: 10.1145/3687299.

[191] *RedMine*. URL: https://www.redmine.org/ (visited on 03/07/2024).

[192] Regnell, B., Berntsson-Svensson, R. and Wnuk, K. 'Can We Beat the Complexity of Very Large-Scale Requirements Engineering?' In: *Requirements Engineering: Foundation for Software Quality, 14th International Working Conference, REFSQ 2008, Montpellier, France, June 16-17, 2008, Proceedings*. Ed. by B. Paech and C. Rolland. Vol. 5025.

Lecture Notes in Computer Science. Springer, 2008, pp. 123–128. DOI: 10.1007/978-3-540-69062-7_11.

[193] Reinartz, W., Krafft, M. and Hoyer, W. D. 'The customer relationship management process: Its measurement and impact on performance'. In: *Journal of marketing research* 41.3 (2004), pp. 293–305.

[194] Rocha, H., Valente, M. T., Marques-Neto, H. and Murphy, G. C. 'An Empirical Study on Recommendations of Similar Bugs'. In: *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*. IEEE Computer Society, 2016, pp. 46–56. DOI: 10.1109/SANER.2016.87.

[195] Runeson, P., Höst, M., Rainer, A. and Regnell, B. *Case Study Research in Software Engineering - Guidelines and Examples*. Wiley, 2012. URL: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118104358.html.

[196] Ruparelia, N. B. 'Software development lifecycle models'. In: *ACM SIGSOFT Software Engineering Notes* 35.3 (2010), pp. 8–13.

[197] Saha, R. K., Lawall, J., Khurshid, S. and Perry, D. E. 'Are These Bugs Really "Normal"?' In: *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. Ed. by M. Di Penta, M. Pinzger and R. Robbes. IEEE Computer Society, 2015, pp. 258–268. DOI: 10.1109/MSR.2015.31.

[198] Saito, S., Iimura, Y., Massey, A. K. and Antón, A. I. 'How Much Undocumented Knowledge is there in Agile Software Development?: Case Study on Industrial Project Using Issue Tracking System and Version Control System'. In: *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*. Ed. by A. Moreira, J. Araújo, J. Hayes and B. Paech. IEEE Computer Society, 2017, pp. 194–203. DOI: 10.1109/RE.2017.33.

[199] Sandusky, R. J., Gasser, L. and Ripoche, G. 'Bug Report Networks: Varieties, Strategies, and Impacts in a F/OSS Development Community'. In: *Proceedings of the 1st International Workshop on Mining Software Repositories, MSR at ICSE 2004, Edinburgh, Scotland, UK, 25th May 2004*. Ed. by A. E. Hassan, R. C. Holt and A. Mockus. 2004, pp. 80–84.

[200] *Savanna*. URL: https://savannah.gnu.org (visited on 16/01/2025).

[201] Scacchi, W. 'Understanding the requirements for developing open source software systems'. In: *IEE Proc. Softw.* 149.1 (2002), pp. 24–39. DOI: 10.1049/IP-SEN:20020202.

[202] Scacchi, W. 'Free/open source software development'. In: *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*. Ed. by I. Crnkovic and A. Bertolino. ACM, 2007, pp. 459–468. DOI: 10.1145/1287624.1287689.

[203] Scacchi, W. 'Free/Open Source Software Development: Recent Research Results and Methods'. In: *Advances in Computers* 69 (2007), pp. 243–295. DOI: 10.1016/S0065-2458(06)69005-0.

[204] Scacchi, W. 'The future of research in free/open source software development'. In: *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. Ed. by G. Roman and K. J. Sullivan. ACM, 2010, pp. 315–320. DOI: 10.1145/1882362.1882427.

[205] Scacchi, W., Feller, J., Fitzgerald, B., Hissam, S. A. and Lakhani, K. R. 'Understanding Free/Open Source Software Development Processes'. In: *Softw. Process. Improv. Pract.* 11.2 (2006), pp. 95–105. DOI: 10.1002/SPIP.255.

[206] Schlutter, A. and Vogelsang, A. 'Knowledge Representation of Requirements Documents Using Natural Language Processing'. In: *Joint Proceedings of REFSQ-2018 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track co-located with the 23rd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2018), Utrecht, The Netherlands, March 19, 2018*. Ed. by K. Schmid. Vol. 2075. CEUR Workshop Proceedings. CEUR-WS.org, 2018. URL: https://ceur-ws.org/Vol-2075/NLP4RE%5C_paper9.pdf.

[207] Seiler, M. and Paech, B. 'Using Tags to Support Feature Management Across Issue Tracking Systems and Version Control Systems'. In: *Requirements Engineering: Foundation for Software Quality*. Ed. by P. Grünbacher and A. Perini. Cham: Springer International Publishing, 2017, pp. 174–180.

[208] Shanks, G. et al. 'Guidelines for conducting positivist case study research in information systems'. In: *Australasian Journal of Information Systems* 10.1 (2002).

[209] Shi, L., Wang, Q. and Li, M. 'Learning from evolution history to predict future requirement changes'. In: *21st IEEE International Requirements Engineering Conference, RE 2013, Rio de Janeiro-RJ, Brazil, July 15-19, 2013*. IEEE Computer Society, 2013, pp. 135–144. DOI: 10.1109/RE.2013.6636713.

[210] Singer, T. and Lamm, C. 'The Social Neuroscience of Empathy'. In: *Annals of the New York Academy of Sciences* 1156 (2009). DOI: 10.1111/j.1749-6632.2009.04418.x.

[211] Sjøberg, D. I., Dyba, T., Anda, B. C. and Hannay, J. E. 'Building theories in software engineering'. In: *Guide to advanced empirical software engineering* (2008), pp. 312–336.

[212] *smartsheet tool.* URL: https://www.smartsheet.com/ (visited on 20/11/2024).

[213] Sommerville, I. 'Software engineering (ed.)' In: *America: Pearson Education Inc* (2011).

[214] *SpiraTest.* URL: https://www.inflectra.com/Products/SpiraTest/Highlights/Bug-Tracking.aspx (visited on 03/07/2024).

[215] Stanik, C., Montgomery, L., Martens, D., Fucci, D. and Maalej, W. 'A Simple NLP-Based Approach to Support Onboarding and Retention in Open Source Communities'. In: *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 2018, pp. 172–182. DOI: 10.1109/ICSME.2018.00027.

[216] Stol, K. and Fitzgerald, B. 'The ABC of Software Engineering Research'. In: *ACM Trans. Softw. Eng. Methodol.* 27.3 (2018), 11:1–11:51. DOI: 10.1145/3241743.

[217] Sutherland J., S. K. *The Scrum Guide—The Definitive Guide to Scrum: The Rules of the Game.* URL: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf (visited on 09/09/2024).

[218] Tamburri, D., Kazman, R. and Fahimi, H. 'The Architect's Role in Community Shepherding'. In: *IEEE Softw.* 33.6 (2016), pp. 70–79. DOI: 10.1109/MS.2016.144.

[219] Tamburri, D., Kruchten, P., Lago, P. and Vliet, H. van. 'Social debt in software engineering: insights from industry'. In: *J. Internet Serv. Appl.* 6.1 (2015), 10:1–10:17. DOI: 10.1186/S13174-015-0024-6.

[220] Tamburri, D., Palomba, F. and Kazman, R. 'Exploring Community Smells in Open-Source: An Automated Approach'. In: *IEEE Trans. Software Eng.* 47.3 (2021), pp. 630–652. DOI: 10.1109/TSE.2019.2901490.

[221] Telemaco, U., Oliveira, T. C., Alencar, P. S. C. and Cowan, D. 'A Catalog of Bad Agile Smells for Agility Assessment'. In: *Proceedings of the XXII Iberoamerican Conference on Software Engineering, CIbSE 2019, La Habana, Cuba, April 22-26, 2019*. Ed. by B. Marín. Curran Associates, 2019, pp. 30–43.

[222] Telemaco, U., Oliveira, T. C., Alencar, P. S. C. and Cowan, D. 'A Catalogue of Agile Smells for Agility Assessment'. In: *IEEE Access* 8 (2020), pp. 79239–79259. DOI: 10.1109/ACCESS.2020.2989106.

[223] Terdchanakul, P., Hata, H., Phannachitta, P. and Matsumoto, K. 'Bug or Not? Bug Report Classification Using N-Gram IDF'. In: *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 534–538. DOI: 10.1109/ICSME.2017.14.

[224] Thayer, R. H. and Yourdon, E. 'Software engineering project management'. In: *Software engineering project management* (1997), pp. 72–104.

[225] Thompson, C. A., Murphy, G. C., Palyart, M. and Gasparic, M. 'How software developers use work breakdown relationships in issue repositories'. In: *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*. Ed. by M. Kim, R. Robbes and C. Bird. ACM, 2016, pp. 281–285. DOI: 10.1145/2901739.2901779.

[226] Tian, Y., Lo, D. and Sun, C. 'Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction'. In: *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*. IEEE Computer Society, 2012, pp. 215–224. DOI: 10.1109/WCRE.2012.31.

[227] Tian, Y., Lo, D. and Sun, C. 'DRONE: Predicting Priority of Reported Bugs by Multi-factor Analysis'. In: *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 2013, pp. 200–209. DOI: 10.1109/ICSM.2013.31.

[228] Tomova, M. T., Rath, M. and Mäder, P. 'Use of trace link types in issue tracking systems'. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. Ed. by M. Chaudron, I. Crnkovic, M. Chechik and M. Harman. ACM, 2018, pp. 181–182. DOI: 10.1145/3183440.3195086.

[229] *Trac*. URL: https://trac.edgewall.org/ (visited on 03/07/2024).

[230] *Trello*. URL: https://trello.com/ (visited on 03/07/2024).

[231] Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., Lucia, A. D. and Poshyvanyk, D. 'An empirical investigation into the nature of test smells'. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by D. Lo, S. Apel and S. Khurshid. ACM, 2016, pp. 4–15. DOI: 10.1145/2970276.2970340.

[232] Tuna, E., Kovalenko, V. and Tüzün, E. 'Bug Tracking Process Smells In Practice'. In: *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 2022, pp. 77–86. DOI: 10.1109/ICSE-SEIP55303.2022.9793952.

[233] van Can, A. and Dalpiaz, F. 'Requirements Information in Backlog Items: Content Analysis'. In: *Requirements Engineering: Foundation for Software Quality - 30th International Working Conference, REFSQ 2024, Winterthur, Switzerland, April 8-11, 2024, Proceedings*. Ed. by D. Mendez and A. Moreira. Vol. 14588. Lecture Notes in Computer Science. Springer, 2024, pp. 305–321. DOI: 10.1007/978-3-031-57327-9_19.

[234] van Can, A. and Dalpiaz, F. 'Locating requirements in backlog items: Content analysis and experiments with large language models'. In: *Information and Software Technology* 179 (2025), p. 107644.

[235] Veitía, F. J. P. *Identifying User Stories in Issues records.* Tech. rep. 2020. DOI: 10.17632/bw9md35c29.2.

[236] Vogelsang, A., Mendez, D. and Franch, X. 'Is RE Research Relevant for Practitioners? First Results from the RE-pract Study'. In: *Softwaretechnik-Trends* 39.1 (2019), pp. 9–10. URL: https://fb-swt.gi.de/fileadmin/FB/SWT/Softwaretechnik-Trends/Verzeichnis/Band%5C_39%5C_Heft%5C_1/04-Vogelsang%5C_et%5C_al.pdf.

[237] Wagner, S., Mendez, D., Felderer, M., Vetrò, A., Kalinowski, M., Wieringa, R. J., Pfahl, D., Conte, T., Christiansson, M., Greer, D., Lassenius, C., Männistö, T., Nayebi, M., Oivo, M., Penzenstadler, B., Prikladnicki, R., Ruhe, G., Schekelmann, A., Sen, S., Spínola, R. O., Tuzcu, A., Vara, J. L. de la and Winkler, D. 'Status Quo in Requirements Engineering: A Theory and a Global Family of Surveys'. In: *ACM Trans. Softw. Eng. Methodol.* 28.2 (2019), 9:1–9:48. DOI: 10.1145/3306607.

[238] Wang, X., Zhang, L., Xie, T., Anvik, J. and Sun, J. 'An approach to detecting duplicate bug reports using natural language and execution information'. In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008.* Ed. by W. Schäfer, M. B. Dwyer and V. Gruhn. ACM, 2008, pp. 461–470. DOI: 10.1145/1368088.1368151.

[239] Wei, L. and Yazdanifard, R. 'The impact of Positive Reinforcement on Employees' Performance in Organizations'. In: *American Journal of Industrial and Business Management* 4 (2014), pp. 9–12. DOI: 10.4236/AJIBM.2014.41002.

[240] Winkler, J. P. and Vogelsang, A. 'Using Tools to Assist Identification of Non-requirements in Requirements Specifications - A Controlled Experiment'. In: *Requirements Engineering: Foundation for Software Quality - 24th International Working Conference, REFSQ 2018, Utrecht, The Netherlands, March 19-22, 2018, Proceedings.* Ed. by E. Kamsties, J. Horkoff and F. Dalpiaz. Vol. 10753. Lecture Notes in Computer Science. Springer, 2018, pp. 57–71. DOI: 10.1007/978-3-319-77243-1_4.

[241] Winter, K., Femmer, H. and Vogelsang, A. 'How Do Quantifiers Affect the Quality of Requirements?' In: *Requirements Engineering: Foundation for Software Quality - 26th International Working Conference, REFSQ 2020, Pisa, Italy, March 24-27, 2020, Proceedings [REFSQ 2020 was postponed].* Ed. by N. H. Madhavji, L. Pasquale, A. Ferrari and S. Gnesi. Vol. 12045. Lecture Notes in Computer Science. Springer, 2020, pp. 3–18. DOI: 10.1007/978-3-030-44429-7_1.

[242] Wisniewski, P. K. and Lu, Y. 'When more is too much: Operationalizing technology overload and exploring its impact on knowledge worker productivity'. In: *Comput. Hum. Behav.* 26.5 (2010), pp. 1061–1072. DOI: 10.1016/J.CHB.2010.03.008.

[243] Xavier, L., Ferreira, F., Brito, R. and Valente, M. T. 'Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems'. In: *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*. Ed. by S. Kim, G. Gousios, S. Nadi and J. Hejderup. ACM, 2020, pp. 137–146. DOI: 10.1145/3379597.3387459.

[244] Xia, X., Lo, D., Wang, X. and Zhou, B. 'Accurate developer recommendation for bug resolution'. In: *20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013*. Ed. by R. Lämmel, R. Oliveto and R. Robbes. IEEE Computer Society, 2013, pp. 72–81. DOI: 10.1109/WCRE.2013.6671282.

[245] Xuan, J., Jiang, H., Ren, Z. and Zou, W. 'Developer prioritization in bug repositories'. In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by M. Glinz, G. C. Murphy and M. Pezzè. IEEE Computer Society, 2012, pp. 25–35. DOI: 10.1109/ICSE.2012.6227209.

[246] Zhang, J., Wang, X., Hao, D., Xie, B., Zhang, L. and Mei, H. 'A survey on bug-report analysis'. In: *Sci. China Inf. Sci.* 58.2 (2015), pp. 1–24. DOI: 10.1007/S11432-014-5241-2.

[247] Zhang, T., Jiang, H., Luo, X. and Chan, A. T. S. 'A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions'. In: *Comput. J.* 59.5 (2016), pp. 741–773. DOI: 10.1093/COMJNL/BXV114.

[248] Zhao, L., Alhoshan, W., Ferrari, A., Letsholo, K. J., Ajagbe, M. A., Chioasca, E. and Batista-Navarro, R. T. 'Natural Language Processing for Requirements Engineering: A Systematic Mapping Study'. In: *ACM Comput. Surv.* 54.3 (2022), 55:1–55:41. DOI: 10.1145/3444689.

[249] Zhou, Y., Tong, Y., Gu, R. and Gall, H. C. 'Combining text mining and data mining for bug report classification'. In: *J. Softw. Evol. Process.* 28.3 (2016), pp. 150–176. DOI: 10.1002/SMR.1770.

[250] Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schröter, A. and Weiss, C. 'What Makes a Good Bug Report?' In: *IEEE Trans. Software Eng.* 36.5 (2010), pp. 618–643. DOI: 10.1109/TSE.2010.63.

[251] Zou, W., Lo, D., Chen, Z., Xia, X., Feng, Y. and Xu, B. 'How Practitioners Perceive Automated Bug Report Management Techniques'. In: *IEEE Trans. Software Eng.* 46.8 (2020), pp. 836–862. DOI: 10.1109/TSE.2018.2870414.

[252] Zowghi, D. '"Affects" of User Involvement in Software Development'. In: *1st International Workshop on Affective Computing for Requirements Engineering, AffectRE at RE 2018, Banff, AB, Canada, August 21, 2018*. Ed. by D. Fucci, N. Novielli and E. Guzman. IEEE, 2018, p. 13. DOI: 10.1109/AFFECTRE.2018.00008.

Term Index

Author Index