FACULTY
OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

University of West Bohemia
Department of Computer Science and Engineering
Univerzitní 8
30614 Plzeň
Czech Republic

# Detecting software development process patterns in project data

The State of the Art and the Concept of Ph.D. Thesis

Petr Pícha

Technical Report No. DCSE/TR-2019-04
August 2019
Distribution: public

## Abstract

Project Management (PM) and Software Process Improvement (SPI) are complex activities demanding decisions which are not clear-cut even when using a defined process based on best practices proven as beneficial to product quality and project success. This is due to specific context surrounding each software development project and the fact that much of the guidance is in textual form not suitable for automatic processing. An deep know-how and extensive manual analysis of the project data and progress is therefore needed to support the correct PM and SPI decisions. This analysis is time and resource consuming, and prone to overlooking important data and reaching incorrect conclusions potentially derailing the project even further.

The goal of this work is to represent the project data using a unified metamodel allowing cross-examination of different projects, detection of defined errors in PM (anti-patterns) as well as points of divergence from the defined software development processes. The expected benefit is streamlining the anti-pattern detection activities, leaving more time to handle their occurrences properly, while minimizing their effects on product quality and project success. To reach the goal we analyzed software development methodologies, Application Lifecycle Management (ALM) tools commonly used in PM, techniques of software process modeling and current research in methods and frameworks to aid PM and SPI efforts. We also review well-known PM anti-patterns from literature and present a method for their operationalization. We then propose an approach to analyze project data from ALM tools and detect anti-pattern occurrence in it. The approach is partially validated through implementation of an experimental tool.

This work was supported by the Ministry of Education, Youth and Sports university specific research project SGS-2016-018 Data and Software Engineering for Advanced Applications.

# Acknowledgements

I would like to express my deep gratitude to my supervisor Doc. Ing. Přemysl Brada, MSc., Ph.D. for continuous support, patience and guidance throughout my studies. I would also like extend my sincere thanks to all the others who supported and helped me along the way, like my colleagues, my friends and my family, and especially, my mother.

# Contents

# Chapter 1

# Introduction

While software development teams have access to large amount of materials and support in terms of guidance and tools surrounding Project Management (PM), they still struggle to implement the context specific best and avoid bad practices (well-defined, frequently occurring mistakes). They could greatly benefit from an automated framework of methods and tools checking compliance of their activities to the specific process they are meant to follow and warning them of the occurrence of suspicious behavior potentially leading to diminishing the product quality or threatening the success of a project.

This work focuses on identification, collection, definition and detection of PM and process patterns (and anti-patterns) in software development projects based on collecting and analyzing their data from Application Lifecycle Management (ALM) tools repositories. Once these patterns are detected (or at least indicated with reasonable probability), there are many opportunities for deriving information from their analysis usable in both further research and industry. This includes studying the correlation between a presence of any pattern and the software product quality, detection anti-patterns (i.e., patterns with detrimental effects on the project or its product), investigation of the compliance with a software process used to the one declared or to a specific methodology, and even predictions about the project's future success/failure based on similar previously analyzed projects.

## 1.1 Problem Definition

### 1.1.1 Software Development Processes and Practices

Every software development project follows some kind of process, a set of methods, practices, activities and tools. Sometimes, a process is well-defined with the intention to be used as a framework to help reach the goals of a project and ensure output quality. This is certainly true of today's highly collaborative, often geographically decentralized and complex projects and even about the simple and one-person projects (though in these projects the process is usually not explicitly described). There is a large body of knowledge in the Software Engineering (SE) discipline regarding processes, methodologies, particular practices, guidance and problem solving approaches. Unfortunately, there are also several major issues in regards to this guidance.

First, because the discipline of SE is still relatively young, there is not exactly a strict and widely accepted consensus on a unified way to capture this knowledge. This results in conflicting terminologies, description format of the practices and processes and opinions on their benefits and pitfalls even in narrowly specified contexts. The second problem is that this guidance exists predominantly as a (structured) description in natural language form not allowing for automated way of use (i.e., operationalization).

These issues result in a state where even if developers and other stakeholders of software development projects know (well or less well) what to do, what to avoid and how to solve eventual problems, there is no straightforward automated mechanism to let them know if they are proceeding as they should (as opposed to how they interpreted the information) and/or if there is a process problem (i.e., anti-pattern) emerging or already in effect. There is a plethora of techniques like mentoring or process auditing and tools that give indication of potential issues, but these are hardly straightforward – the former being expensive and time-wise inefficient, and the latter often decentralized (dependent on checking different aspects of a project in various tools) and in need of a personal deductive skills, instincts and experience of the person examining the data. Therefore, a deviation from the course or a problem occurrence is detected late, if ever, making the course correction harder or impossible and leading to project being overdue, over budget or failing completely. Moreover, making this process automated and data based would make it faster, repeatable, data-based, as objective as possible and mitigate the human or judgment error proneness. In the meantime, it can still be configurable for specific needs of each particular project and its context.

### 1.1.2 ALM Tools and Their Data

On the other hand, with the ever-growing complexity of the software projects, involving more people geographically further away from each other, building larger software, inter-application integration, etc., the usage of tools to manage these projects has become practically a necessity. Consider the Open-Source Software (OSS) development projects. These could hardly even exist without the support of tools for managing requirements, sharing code, individual versions, change requests, testing, defect reports and fixing. And the in-house projects complexity makes it hard to imagine for them to function without these supporting tools as well. So once again, virtually every software development project utilizes one or more of so called ALM tools. This group of tools includes applications supporting every phase and discipline present in the projects like Version Control System (VCS), ticketing (a.k.a. issue-tracking, or change management) systems, requirements management systems, quality management systems for testing, etc. Tools like Git, GitHub, Apache Subversion (SVN) and Atlassian Jira gained widespread popularity and are commonly used in practice.

As a result, the repositories of these tools contain large amounts of data about projects and (if handled honestly and properly, see section 3.4.3) the accurate account of their history and present reality. There is a great amount of potential in analyzing this data and deriving further information about the projects. And although some tools provide means of simple analysis like filers and charts, there is still untapped potential. Examples of such potential include cross-examination of different projects, not to mention from different source tools, and complex analyses giving clear indication on (anti-)patterns presence/absence and process compliance. Our proposed automated approach to collect and analyze the data form ALM tools can harness this potential and benefit the project staff with the contextually relevant and concrete data-based information to base the PM and Software Process Improvement (SPI) decisions upon.

## 1.2 Goal of the Work

To summarize the previous section, there is an abundance of methodological guidance which people struggle to utilize on one hand, and support from tools describing in detail the reality of the projects on the other, present in a large majority of collaborative software development efforts. This thesis proposes utilizing data collection and analytics techniques applied to the project data obtained from ALM tools to gain information regarding the compliance of a specific project with a defined process and to identify well-known, frequently occurring mistakes (anti-patterns) in PM.

We expect the process patterns can be detected, or at least indicated, automatically and with sufficient reliability from data collected form ALM tool repositories with little necessity for extra input form the project staff. Therefore the hypothesis of our work is as follows:

**Hypothesis**: *The occurrence of PM (anti-)patterns can be automatically detected in ALM data of a given project.*

Each research question (RQ) from the following set must be answered affirmatively to except this hypothesis.

- **RQ1**: *Is it possible to create an unified generalized process representation capturing essential elements and semantics of various process models and suitable for storing project data mined from ALM tools?*
- **RQ2**: *Can PM patterns and anti-patterns be represented in an operationalized form needed for their automatic detection?*
- **RQ3**: *Do patterns representing practices and processes occur in the ALM data?*
- **RQ4**: *Does the presence or absence of patterns and anti-patterns have a relation to the product quality and project success?*

If the hypothesis is accepted based on these RQs a gap between methodological knowledge and actual project data can be bridged. A framework can be built to analyze project data, detect (anti-)patterns, inform the developers and thus help them make timely, data-based PM and SPI decision. This would then lead to better product quality, increase in project success chances, lower costs on manual or outsourced process audits as well as on consequences of bad decisions and late or no course corrections in PM and SPI.

Similar automated approaches already exist on project elements, such as coding and design patterns. Inspiration can be drawn from these techniques, from generic pattern description and detection methods and even from existing approaches similarly focused to ours. Nevertheless, the systematic and operationalized mechanism for process pattern detection and pattern form suitable for such processing is missing in state-of-the-art research. This is, however, vital to objective studies of the effects of (among others) anti-pattern occurrence and process deviation on project success and product quality. Our focus is therefore strictly on the process and PM patterns, specifically anti-patterns (patterns with harmful effects on project and product).

The text of this report first describes the context of the work, including key concepts, terminology, software development methodologies and ALM tools (Chapter 2). Then the review of similar and related work from areas of Mining Software Repositories (MSR), pattern detection and project and product measuring is presented (see Section 2.4). After that, the concept of our approach and current state of our research is described (Chapter 3) followed by discussion of future directions of our efforts (Chapter 4) and conclusion (Chapter 5).

# Chapter 2

# Background and Related Work

This chapter presents the essential context of and terminology used throughout the work as well as efforts already undertaken by other researchers in the fields related to our goals.

## 2.1 Key concepts

This section describes the key concepts our work builds upon. The sources of terminology and definitions are, among others, well-established metamodels like Software & Systems Process Engineering Meta-Model ([86], see Section2.5.1) and Open Services for Lifecycle Collaboration ([89], see Section 2.5.2), and methodologies like Unified Process (see Section 2.2.2) and Scrum ([112], see Section 2.2.3).

### 2.1.1 Process

A software development process is a set of interrelated and time-wise ordered (though not necessarily always sequential) activities, their designated performers, inputs and outputs leading to a common goal of developing a software product through multiple stages and intermediate milestones. A process can be also decomposed to sub-processes making it a recursive concept.

Three main building blocks of processes are:

- **Roles** – Sets of competencies and responsibilities usually associated with a particular activity (e.g., requirements analyst, developer, systems tester, etc.) assigned to a person. A role-person relation is of N:M cardinality, meaning a person can assume one or more roles in the span of a project and one role can be assumed by multiple people.

- **Tasks** – Atomic units of work effort with specific goal, inputs, outputs and place in the workflow of the process undertaken by one or more people of one or more roles (e.g., describe requirements, unit-test particular functionality, perform a team meeting).
- **Work Products** – Inputs and outputs of tasks, activities or the whole process. The level of ceremony and concrete form can vary vastly from formal rigorous documents with detailed templates (e.g., Software Requirements Specification document), source code itself, test cases (written scenarios, automatized scripts, etc.), plans in various forms, to sketches, hand-written notes, screenshots, or even just a certain knowledge held in minds of the developers with no physical form whatsoever. In wide SE practice a term artifact is more commonly used for the concept. In [86] the term *Artifacts* is used just for a subset of *Work Products*, with *Deliverables* and *Outcomes* being the other subcategories.

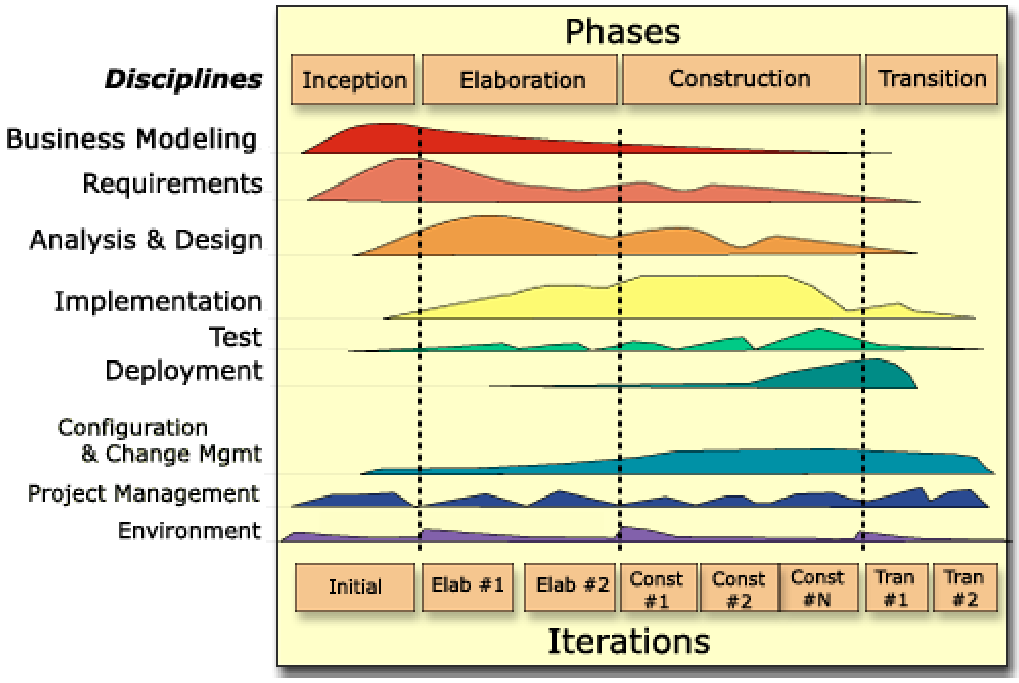


Figure 2.1: Unified Process phases and disciplines [68]

With the growing complexity of processes, other concepts come into play, like:

- **Activity** – A set of related tasks with a common specific goal, or a periodical occurrence of a specific task (e.g., communication with a customer).
- **(Sub-)Process** – The larger the process, the bigger the need to decompose it. As mention above, the overall process may be divided into

smaller, partial (sub-)processes that allow focus on the tasks at hand leading to intermediate milestone. This includes phases (in sequential and iterative methodologies), iterations (time-wise smaller repeatable instances of the project lifecycle in iterative, agile and lean methodologies), weekly, and even daily processes.

- **Guidance** – Anything supporting, easing and automating the process. Guidance includes artifact templates and examples, guidelines, manuals and tutorials for tools, practice and method descriptions, supporting literature, knowledge base, experience reports from previous projects, roadmaps, whitepapers, training materials, tools for various project activities (communication, configuration and change management, task assignments, planning, issue-tracking, etc.).

- **Category** – Other concepts (roles, tasks, artifacts, guidance, etc.) can be grouped into categories based on their various aspects. Example of such sorting are disciplines, dividing the process content by focus on particular domain (e.g., design, implementation, testing, deployment, etc.). This categorization may correspond with phases (e.g., in sequential methodologies) or be orthogonal to them (e.g., iterative approaches; see Figure 2.1). Each task and artifact is usually associated with one or more disciplines.

### 2.1.2 Methodology

Software development methodologies are authored, well-described and semi-standardized process models. Apart from the process model itself a guidance on the its usage, adoption, scaling and tailoring, etc. is usually included in the methodology, as well as the history of its evolution and justification for its creation. Some methodologies even include several variations of the process model (i.e., its overall workflow) for greater usability and context specific settings of the project. Methodologies are predominantly a collection of best practices and methods leading to the successful completion of the software project.

### 2.1.3 Project

A software project is an executed instance of the software process. It is set in a specific context, has concrete resources and goals. It can go through either only some of the workflow branches of the process, or all of them based on its context. Projects also have the ability to fine-tune their activities according to their context and needs, because of the different levels detail of the process and methodology used, as mentioned above. In some cases this can even mean deviating from the given process model by incorporating practices not covered by the description (or prescription) of the model, or ignoring those included.

### 2.1.4 Practice

> *"A practice represents a proven way or strategy of doing work to achieve a goal that has a positive impact on work product or process quality. Practices are defined orthogonal to methods and processes. They could summarize aspects that impact many different parts of a method or specific processes."*[86]

Some sub-processes or ways of performing a specific task (e.g., producing a certain artifact) are generalized and partially standardized, or at least anchored in commonly used terminology. These are called practices.

Practices can describe which actor with which role performs which task in which way using which tool or technology on which input to produce which output, or be more general and abstract. Some practices can borderline on something more like approaches permeating through the whole process (e.g., iterative development), some can be as specific as using a particular technology to capture concrete knowledge, for example, Unified Modeling Language (UML) class diagram). Practices can encompass one or more tasks with various level of detail and their specific context and concepts related to them. Furthermore, practices can be extracted from processes and used in other processes, making them the building blocks of processes. The process can really be viewed as a set of practices.

Like processes, practices too can be adjusted by their specific context, specified to various level of detail, and be recursive (i.e., composed of other practices). A practice can be viewed as a sub-process template, or pattern, and, conversely, a process or methodology can be viewed as a high-level and complex practice encompassing the whole project.

### 2.1.5 Pattern and Anti-pattern

The Merriam-Webster dictionary[1] provides several definitions of a pattern. Among others:

- a form or model proposed for imitation,
- something designed or used as a model for making things,

[1] https://www.merriam-webster.com/dictionary

- a natural or chance configuration,
- a reliable sample of traits, acts, tendencies, or other observable characteristics of a person, group, or institution,
- a discernible coherent system based on the intended interrelationship of component parts,
- frequent or widespread incidence.

The term has been adapted to computer science from the work of architect Christopher Alexander [1], who defined design pattern as *“re-usable form of a solution to a design problem”*.

In SE context, the term pattern can have multiple meanings:

1. In a broad sense, pattern is any identifiable and reusable concept like data structure, practice, process component, behavior, etc. irrespective of judgment on benefits or detriments of its occurrence. Specifically, process pattern is then defined in [86]:

    > *“Process Pattern is a special Process that describes a reusable cluster of Activities in a general process area that provides a consistent development approach to common problems.”*

2. Pattern is a commonly known and occurring template, which has positive consequences on the situation. In the realm of software processes and PM, these are also called good or best practices. In sources like [17] and [71] the term pattern is used in an even narrower meaning. Here a pattern is a tested and reusable correct solution to a well-known problem.

Anti-patterns are *“common approaches to solving recurring problems that prove to be ineffective”* [2]. Therefore, anti-patterns can be viewed as a subset of the first meaning of patterns, or an opposite to the second meaning of patterns. They are also referred to as bad practices or smells (e.g., [91]).

In context of our work we will use the first, broader definition of a pattern. Because our work is focused on patterns in general and anti-patterns specifically, we have little use for a term to describe solely beneficial patterns. Nevertheless, our approach can certainly be used for their handling as well, since they are just another specific subset of patterns.

Both patterns and anti-patterns exist on various levels in software development, like code, design, architecture, community, organization, environment, collaboration, etc. This work focuses predominantly on the class of PM (anti-)patterns. In that context, anti-pattern can be an emerging problem, a mistake made, poorly applied solution to a problem, misused best practice or a deviation from a prescribed/recommended process.

## 2.2 Software Development Methodologies

As the best practices and methods had to evolve with time and changing demands on the projects, so had the methodologies themselves evolve. To put it simply, what was once a perfectly manageable strategy for a software project in the era of standalone and (by today's standards) simple applications specific to scientific calculations, is insufficient now in the times of huge, heavily interconnected, multipurpose systems developed by hundreds of people (or more) over several years and multiple platforms, utilizing many different technologies and servicing potentially millions of users.

Some methodologies were not created in the software development domain but rather adopted from manufacturing or other industries. For example, the authors of [114] introduced the term Scrum (more in Section 2.2.3) in 1986 based on case studies from manufacturing firms in the automotive, photocopier and printer industries. And Kanban (more in Section 2.2.3) was inspired by the Toyota Production System[88].

Methodologies are most frequently categorized based on one common aspect of their approach but the categorization schema also correlates with their age. The methodologies can be categorized into ad-hoc, sequential (also called traditional), iterative (or incremental), agile, lean, hybrid (amalgamations of several different types of methodologies), etc. The classification may vary depending on the source as there is still a dispute on e.g., whether lean methodologies are a subset of agile, which in turn can be viewed as a subset of iterative methodologies.

The purpose of this work is not to give the full classification and descriptions of different methodologies. The following subsections only describe a few well-known process models in regards to some of the practices (in bold) that distinguish them from each other and present potential patterns to look for in the project data. If detected, these could be used to identify a process model used by the project or to check the project's adherence to the declared process.

### 2.2.1 Waterfall

The most well-known example of the sequential methodologies is the Waterfall model [101]. Despite many disadvantages of its sequential approach, especially for modern dynamic projects, the Waterfall is used in practice to this day in projects, where frequent changes are not expected, legally possible, or for legacy reasons.

The practices used include:

- **Phases** – The phases divide the lifecycle, have strict focus on one aspect of the project, are sequential with no overlap and scarcely (only if necessary) go back up the workflow seen in Figure 2.2. Once the precise number and focus of the phases is known, this division itself can be considered a pattern and signs of it can be investigated.
- **Phase content** – Each of the phases includes particular tasks and has several outputs to be expected on their conclusion. Furthermore, in strict application of the sequential approach, once created, the artifacts should become static and not be amended in later phases. The artifacts and activities can be specified as patterns and their presence verified.
- **Role activity** – Similar to artifacts being created and finalized in particular phases, personnel with a particular role has its place only in some phases. For example a person with only the *Developer* role should not be active in the *Analysis* phase. This presents a potential pattern to look out for.
- **Artifact form(ality)** – For the transition between two phases to be executed smoothly, the ending phase has to produce artifacts of the highest level of detail and content volume possible for the starting phase to be able to build upon them. Therefore the pattern detection can include measurements of size, structure and content of artifacts.

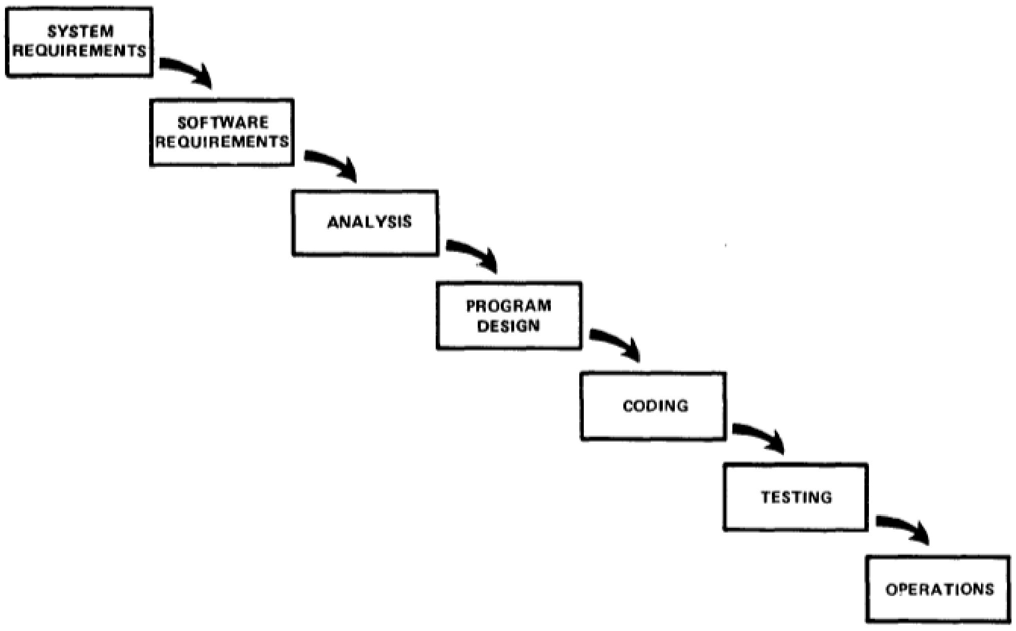


Figure 2.2: Waterfall process model [101]

### 2.2.2 Unified Process

Perhaps the most well-known and commonly used example of the iterative methodologies is the Unified Process (UP), from which more specific and proprietary forms, such as IBM Rational Unified Process (RUP) [68], are derived. UP specifies nine disciplines (see Figure 2.1) and corresponding roles like **Analyst**, **Architect**, **Developer**, **Tester**, **Project Manager**, etc.

The process of UP is divided into **four phases**, each of which is concluded when a specific milestone is reached. A milestone is a set of criteria on the state of the product and the knowledge among the staff vital to the continuation of the project. Since they represent the partial state of the product (code and other artifacts), the activities leading to that state and knowledge acquisition (ideally captured in other artifacts), the **criteria of milestones** can be viewed as patterns and described as such. Phases themselves, their content and activity of appropriate roles can also serve as detectable patterns similarly to the Waterfall model.

Moreover, being a representative of iterative methodologies, the UP process is divided into time-constrained repeatable sub-processes, known as iterations. Each of these has a basic structure of:

- **Planning meeting** – includes selecting tasks to be performed during the iteration, their prioritization and assigning to team members,
- **Iteration execution** – depending on the phase and overall position of the iteration inside the project lifecycle, but generally including activities from all the disciplines,
- **Customer feedback** – a meeting with a customer to showcase results of the iteration and decide on the next steps,
- **Iteration review** – contemplating the leftover work (if any exists), the success of the iteration and possible approach adjustments.

The patterns related to iterations can include their number, length, structure, presence of the appropriate activities and outputs, etc.

On top of the phases and iterations, UP puts great emphasis on practices, like **use-case driven development**, early **baselining and testing of** the potential **architecture**, and continuous **risk management**, among others. Each of these can be conceptualized as a pattern (or set of patterns), for example, creation and updates of use-case models and descriptions, presence of design and testing activities early in the process, and risk list existence and maintenance throughout the lifecycle, respectively. The same can be done with other practices as well.

Apart from RUP, UP also spawned other related process models, like Enterprise Unified Process (EUP) [7] and OpenUP [69]. EUP enhances the RUP model with new phases *Production* and *Retirement* to cover the whole product lifecycle, and 8 new enterprise disciplines on top of the development ones from UP (see Figure 2.3). They can be translated into patterns in similar fashion as practices from UP.

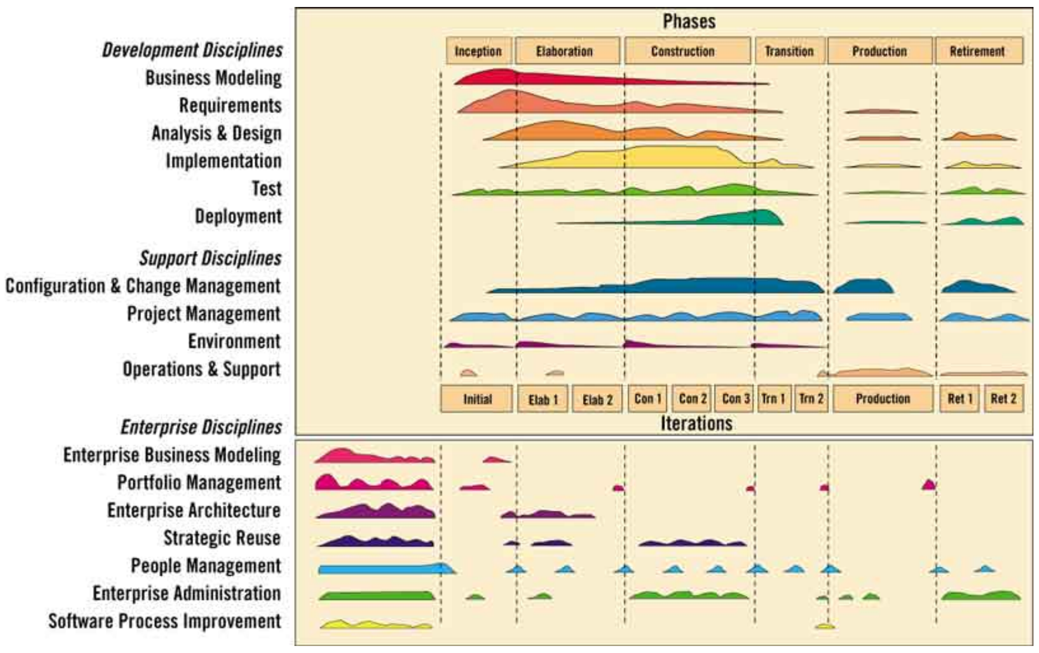


Figure 2.3: Enterprise Unified Process phases and disciplines [7]

OpenUP can be viewed as a lightweight and more agile form of UP contrasting with robust and well-defined RUP. It puts smaller emphasis on tools and artifact formality, introduces microincrements (small increases of functionality delivered after each iteration) and self-organizing teams. OpenUP is an open-source approach constantly evolving by contributions of a wide community. It specifically outlines and describes practices used making it easier to define the patterns to look for.

## 2.2.3 Scrum

In 2001, a group of practitioners trying to push development of so called lightweight methodologies, authored the Agile Manifesto [12].It centering on a set of four core values:

- 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 manifesto became a basis for agile methodologies. The principles of the agile methodologies (such as putting the flexibility for dealing with changing requirements at the forefront, and delivering potentially shippable increments in functionality regularly) have existed and were used long before the actual creation of the document itself.

The proof of this is the most commonly known representative of agile approaches, Scrum, which was presented in 1995 by Schwaber and Sutherland [112]. Due to frequent misinterpretation of its core principles in practice, Scrum can simultaneously be a perfect candidate for our intended process adherence checking, and a difficult process to detect. In accordance to the Agile Manifesto, the practices are just a set of well-established and tested means of guiding the process to maximize its output and quality that proved to benefit most projects. They are not prescribed dogmas demanding the developers follow them at all cost. The overall process model is shown in Figure 2.4.

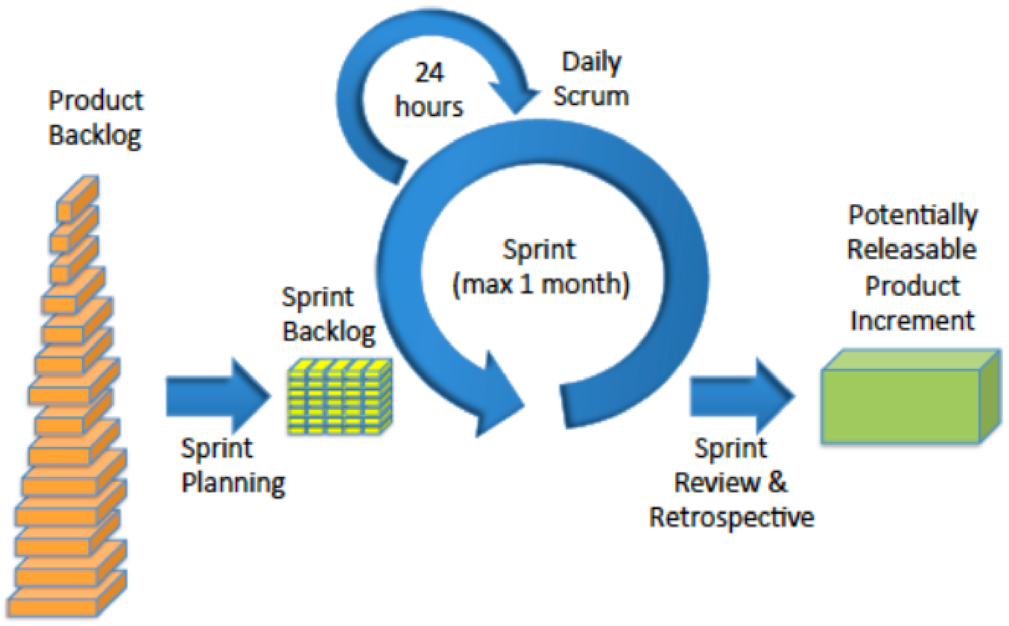


Figure 2.4: Scrum process model [37]

Among the core practices of Scrum are:

- **Self-organizing Teams** – No external structure, hierarchy or role assignment is imposed on the team. And though it might exist internally, to the people outside the team all its members have seemingly the same role, taking collective responsibility for their outputs. Apart from the actual role name (i.e. *Team Member* instead of e.g. *Devel-*

*oper*) the pattern can be identified through the activity of each team member being discipline-wise non-specific, meaning everyone can have an area of dominance, but participates in tasks from all disciplines. This is not only encouraged by Scrum, but often a natural result of the self-organization.

- **Sprint** – *Sprint* is a Scrum term for iteration, most commonly 30 days long (or shorter). It is an uninterrupted period of development efforts between two subsequent meetings with a customer. The meetings themselves then serve two main purposes: 1) for the development team to showcase the results of the *Sprint* (a.k.a. **Sprint Demo**), 2) to reach an agreement with the customer on the following steps to be taken during the next *Sprint* (a.k.a. **Sprint Planning Meeting**). The latter includes deciding on the priorities of current unsatisfied requirements and even their eventual changes and allocating the available resources to gauge the amount of progress attainable, to which the team commits.

- **Backlog** – *Backlog* is a form of a plan. It consists of the items to be processed (features to be implemented) and is usually prioritized, though the set of requirements and their priorities are constantly a subject to change. Usually, two forms of a *backlog* exist. One is for the current iteration consisting of well-described, well-decomposed tasks, which is essential for their most accurate estimation. The other is a project backlog including all the yet-to-be-done tasks in the whole project. These can be in different levels of decomposition, detail of description and estimation accuracy based on their priorities and current knowledge about the project.

- **User Stories** – Short descriptions of fine-grained functional requirements written from the viewpoint of the actor (most commonly user) using a particular functionality. They are usually captured on sticker notes on so called *Agile Board*, physical or virtual (in a visual software tool). The precise format of the board is not standardized, but usually has a table with a row per each potential assignee, a column per each state in the task workflow and description and effort estimation on each note. The unassigned tasks are situated outside the table border usually on the left-hand side (as the state columns go logically from left to right). Through moving the stories through the table on the board, everyone can get an easy and quick overview of the project status and tasks currently processed by each individual team member.

- **Daily Scrum** – A short daily meeting of the development team serving as a synchronization point to inform one another about the current situation, tasks in progress, effort distribution and potential issues.

- **Sprint Retrospective** – An internal meeting of the development team usually shortly after a *Sprint* ended. It serves a similar purpose as *Daily Scrum* but on the *Sprint* scale. Here, the progress and project status is discussed, the process issues are addressed and changes to the process made, if necessary.
- **Timeboxing** – A practice of putting a strict time constraint on some activities. The intend is to keep focus on the important topics and not to get sidetracked or bugged down by too much inconsequential details. This practice is usually applied to *Sprints*, *Daily Scrum* and other meetings, both internal or with the customer.
- **Specific Roles** – Apart from the **development team**, which can be further differentiated internally but is viewed as a coherent whole from the outside, Scrum defines two other major roles. **Product Owner** represents the stakeholders and has the responsibility to keep the team on track in terms of the business case. **Scrum Master**, on the other hand, keeps an eye on the proper use and adherence to the Scrum process. He advises the team on possible and proper enhancements to the process in accordance with the context and status of the project, enforces established policies (e.g., time constraints on meetings, etc.) and keeps the team focused on the important practices. Both roles can be assumed by personnel outside of the team (but within the organization), individuals outside the team's organization and even team members themselves, though the accumulation of both roles in one person is mostly viewed as a bad practice.

Scaled Agile Framework (SAFe) (currently in version 4.5) [103], Nexus [15] and Large Scale Scrum (LeSS) [118] are relatively new agile methodologies, trying to address the scalability issue that many practitioners still seem to have with Scrum and other agile approaches. Their practices are in essence the same as in Scrum, but extended to multiple teams and focusing on "the bigger picture". The adjustment of the patterns is then mostly a calibration issue.

### 2.2.4 Extreme Programming

eXtreme Programming (XP) [13] is based on the idea of taking the beneficial practices from traditional SE to the extreme. Figure 2.5 shows the core practices and their relations.

Some of the practices are easily detectable as patterns:

- **On-Site Customer** – a person with a customer role involved in day-to-day activities;

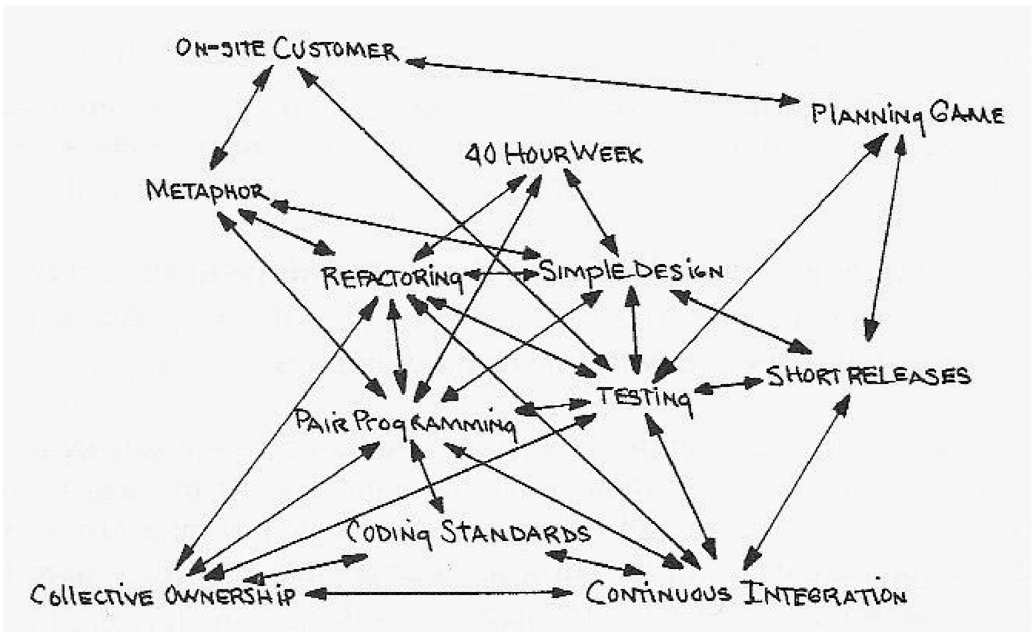


Figure 2.5: Extreme Programming practices [13]

- **40 Hour Week** – strictly timeboxed amount of effort a developer can put in on any given week detected by measuring estimates of assigned work and spent time reported;
- **Refactoring** – changing source code without changing its functionality for better comprehensibility, modularity, reusability, etc; detectable by categorization or labeling of tasks or analysis of associated changes in the source code;
- **Testing** – the presence and prevalence of testing activities, for instance, unit testing of all code;
- **Pair Programming** – coding in pairs of one developer typing the code, the other instantly checking it; done for purposes of immediate *code review* and consultation, exchange of knowledge and experiences, etc.; detectable by similar spent time for similar time period reported by two developers on the same task;
- **Short Releases** – short periods of time between two subsequent releases
- **Coding Standards** – guidelines on structuring, formatting, commenting, committing, etc. of produced source code; detectable by the existence of artifact holding the coding policies of their inference from the code itself;

- **Continuous Integration** – newly developed features are immediately integrated into the overall product and tested in its context; detectable by monitoring integration and testing activities, and measuring time to integration of a newly introduced feature.

Other practices of XP include avoiding programming of **features** until **just-in-time (JIT)**[2], a **flat management structure**, **code simplicity** and clarity, expecting **changes in requirements** with time and increasing comprehension of the problem, and **frequent communication** with the customer and among programmers. All of those are apparently decomposable to aspects detectable through patterns.

### 2.2.5 Kanban

Stemming from David Anderson's 2010 book of the same name [8], Kanban is a representative of the lean approaches. Lean Software Development (LSD) came into existence as a subset of the agile community adopting the principles of lean manufacturing as seen in Toyota Production Systems. It takes almost all the concepts from agile and pushes them to even further agility. The principles of LSD are:

- **Eliminate waste** – including extra processes and features, waiting periods, defects and management activities;
- **Amplify learning** – focus on knowledge and experience gaining and reuse;
- **Decide as late as possible** – pushing decisions to JIT moments, after most of the relevant information comes to light;
- **Deliver as fast as possible** – allows for gaining quick approval and feedback from the customer and minimizing misunderstandings and time spent on the wrong, unwanted or extra features;
- **Empower the team** – give as much liberties and responsibilities to the team as possible to foster sense of ownership and feeling of personal investment;
- **Build integrity in** – meaning to build it into the system through immediate testing, refactoring and architectural principles for modularity and reuse;
- **See the whole** – "think big, act small, fail fast, learn rapidly".

[2]i.e., until the moment they are actually needed

Kanban practices differentiating it from agile methodologies like Scrum, and therefore making the approach detectable include **no prescribed roles**, **continuous delivery** of each feature as soon as it is developed and tested – as opposed to batch delivery each *Sprint*, **changes allowed anytime** – as opposed to no changes allowed mid-sprint, high degree of **variability in priority**, **limiting work in progress**, making **policies explicit**, visualization of progress through **Kanban Board** (similar to *Agile Board*) presenting the requirements in a form of user stories and showing their state in the workflow, **descriptions** of the features **just detailed enough** to eliminate potential misunderstandings, **frequent feedback loops** with customer – as opposed to meetings at *Sprint* edges.

### 2.2.6 Disciplined Agile Delivery

In 2012, Ambler and Lines proposed small changes to the Agile Manifesto along with a new methodology called Disciplined Agile Delivery (DAD) [6]. The methodology is a hybrid, taking practices from previously existing approaches and melting them together to create still agile, but more structured, defined and scalable alternative to Scrum. It adopts practices from:

- **Scrum** – priority based processing of tasks, product owner, potentially shippable product delivered after each *Sprint*, etc.
- **RUP** – continuous integration, refactoring, test-driven development, collective ownership, etc.
- **Agile Modeling** – continuous documentation, requirements and architecture envisioning, iteration modeling, JIT model storming, etc.
- **UP** – “lightweight” milestones, explicit phases, focus on baselining the architecture early, risk mitigation in early stages of the lifecycle, etc.
- **Agile Data** – database refactoring, database testing, agile data modeling, agile enterprise strategy, etc.
- **Kanban** – limiting work in progress, visualizing workflow, etc.

The process is divided into phases similar to UP, except for the elimination of *Elaboration*, from which most of the activities were moved to *Construction*. The approach makes use of agile practices like *Daily Coordination Meeting*, *Backlog*, *Demo* and feedback gathering. The structure of the whole process is visible in Figure 2.6.

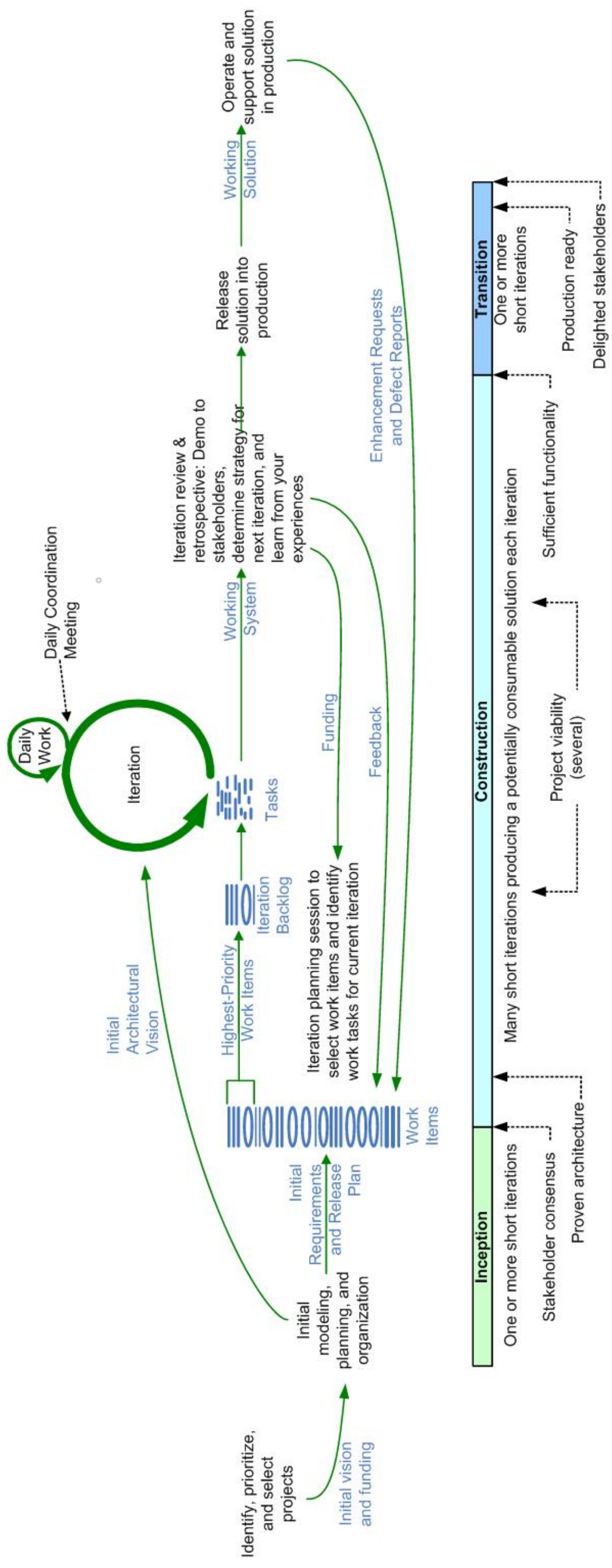

Identify, prioritize, and select projects
Initial vision and funding
Initial modeling, planning, and organization
Initial Requirements and Release Plan
Work Items
Initial Architectural Vision
Highest-Priority Work Items
Iteration Backlog
Tasks
Iteration planning session to select work items and identify work tasks for current iteration
Funding
Feedback
Iteration
Daily Work
Daily Coordination Meeting
Working System
Iteration review & retrospective: Demo to stakeholders, determine strategy for next iteration, and learn from your experiences
Enhancement Requests and Defect Reports
Release solution into production
Working Solution
Operate and support solution in production
Inception
One or more short iterations
Stakeholder consensus
Proven architecture
Construction
Many short iterations producing a potentially consumable solution each iteration
Project viability (several)
Sufficient functionality
Transition
One or more short iterations
Production ready
Delighted stakeholders

Figure 2.6: Disciplined Agile Delivery process model [6]

### 2.2.7 DevOps

With the demands on ever-faster time-to-delivery and factoring in the nature of most modern software systems, where development of a new release coincides with operations and maintenance of the previous one, while extensive down-time due to new release deployment is deemed undesirable, a new approach to satisfy these needs was created. The approach is called DevOps (a clipped compound of the words "Development" and "Operations"), a term steadily used since 2009. It is perhaps best summarized in the definition proposed by Bass, Weber, and Zhu [11]:

> *"DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality."*

DevOps rose from the success of agile approaches and pushes the boundaries of their philosophy even further through continuous development, integration and delivery. The point of the approach is to effectively chain the activities of coding, validation (*code review*), building, testing, packaging, release, (infrastructure) configuration and monitoring (of performance, end-user experience, etc.). Adoption of DevOps in an organization requires a shift in organizational and cultural paradigms, mainly because of the conflicting nature of department roles of Operations, Developers and Testers, which now need to work together. The issue is that the goals of the roles (organizational stability, change, and risk reduction, respectively) often contradict each other and achieving cohesive cooperation is therefore one of the main challenges of the DevOps approach. All of the above mentioned practices can be considered patterns, and especially the latter described challenge of collaboration between roles presents potential for anti-pattern occurrences.

## 2.3 ALM tools

As discussed in introduction, the methodologies, processes, practices and patterns and their descriptions are just one of the two major parts this work tries to bring together. The other part are ALM tools.

ALM tools are a group of software means supporting the development of an application from the very conception of an idea in the business domain to development, deployment, operations, maintenance, all the way to retirement and decommission. They allow the development teams and other stakeholders to track the progress of the software projects, see who and when made which changes to the artifacts pertaining to particular activity, archive communications for later references, document the day-to-day events, report defects, build plans and manage the projects sources and goals to maximize the satisfaction on all sides.

The scope of their usage in a project may vary based on parameters such as number of people involved, project scope, timespan, geographical collocation of the team members, legal requirements (e.g., some standards demand the records of the development to be kept for a certain period of time), etc. Apparently, a week long project with one developer delivering a simple utility to a colleague sitting in the same office will call for much less involvement of ALM tools, which would only add to unnecessary administrative overhead. On the other hand, a multi-year project developing a new commercial operating system involving collaboration of hundreds of programmers from several teams spread out throughout the globe obviously merits the utilization of tools for the project to even be manageable. This subsection describes several ALM tool categories of particular interest in regards to our data collecting and analysis needs as a part of our work.

### 2.3.1 Version Control Systems

VCS are tools to keep track of the code (and other artifact) changes. They keep the record of who and when made which exact change to which artifact stored in their repository. They are used in situations where multiple people need to simultaneously work on common set of artifacts (software, documents, etc.) without creating conflicting changes and with minimal time waste.

Each of the collaborators has a private (working) copy of the repository content which needs to be regularly brought up to date (synchronized with the common repository). They then apply changes to the artifacts locally, upload (commit) the changes to the main repository, and deal with potential conflicts. Conflicts occur when there are changes on the same artifacts done by another collaborator in between the time that the first one updated his working copy and committed his updates. For each change, the tools record the changes to the artifacts themselves, timestamp, the author of the change, a unique identifier of the change and a message from the collaborator describing the change. Thus, they allow the collaborators to return to any of the previous versions of the artifact or overall state of the repository. This is just the general purpose description, but VCS tools tend to have more functionality, such as tagging versions, multiple development branches, patching, reviews, forking repositories, etc.

Among the best known VCS tools are Git[3], Apache SVN[4], Concurrent Ver-

[3]https://git-scm.com/

[4]https://subversion.apache.org/

sion System (CVS)[5], Mercurial[6] or Atlassian Bitbucket[7].

### 2.3.2 Issue-tracking Systems

Issue-tracking systems (also known as issue-trackers or ticketing systems) are a generalized version of bug-trackers serving predominantly the needs of software change and PM. Their main usage is to log in planned tasks, defect (bug) reports, and other work to be done in tickets. Tickets are forms where all the parameters of the task are captured. This includes a short summary, generated identifier, comprehensive description, priority, due date, an assignee to perform the task, estimated and actual effort to finish the task, current status of the task in a specified workflow, associated commits from VCS, etc. This supports project activities like planning, change management, progress monitoring, and tracking of requirements form their specification to implementation and testing. These activities are further supported by additional features of several issue-trackers, like release planning, relating tickets to one another, progress visualization through plans, reports and charts, time tracking, etc. Some issue-trackers can even be integrated with instances of certain VCS tools completing the commit-issue traceability with interactive features. The aforementioned bug-trackers are systems of the same basic functionality but solely focused, at least in their early existence, on reporting defects and their tracking through to fixing.

The commonly used examples include GithHub[8], Redmine[9], Mozilla Bugzilla[10], Assembla[11], IBM Rational Team Concert (RTC)[12] or Flyspray[13]. The aforementioned bug-trackers (e.g., Bugzilla and Flyspray) are systems of the same basic functionality but solely focused, at least in their early existence, on reporting defects and their tracking through to fixing. Over time they became issue-trackers either through their evolution, or the manner in which developers use them. Different tools also differ in the extensiveness of their features and the detail of information they are able to capture. For instance, GitHub, a tool massively used in OSS development, captures in ticket only the ID, summary, description, assignee, project, comments and simple open/closed status. To add further information, like priority, task type (defect, feature, task, etc.), resolution, approvals and relation to commits or

[5]https://www.cvs.com/
[6]https://www.mercurial-scm.org/
[7]https://www.bitbucket.org/
[8]https://github.com/
[9]https://www.redmine.org/
[10]https://www.bugzilla.org/
[11]https://www.assembla.com/home
[12]https://www.ibm.com/us-en/marketplace/change-and-configuration-management
[13]https://www.flyspray.org/

other tasks is realized through either tags or links in the text descriptions and comments. The terms "issue" or "task" can be sometimes used in place of "ticket". We tend to avoid this terminology as some tools use these words as a type (sub-class) of tickets. Therefore, for the abstract concept in process we use "task" and for its representation in a issue-tracking tool we tend to use "ticket".

### 2.3.3 Knowledge Bases Software

Though certainly not beyond the realm of their abilities, VCS tools are not always used to store and manage all of the project artifacts. It may be the case that the development team uses VCS to manage only code, configuration files, build scripts and other utilities necessary to run the application. Or maybe they additionally keep only the artifacts meant for the customer (deliverables) in VCS, but internal project documentation and knowledge base is kept elsewhere. Some issue-tracking systems have features, modules or plug-ins for wiki pages or document versioning, but the knowledge basis storage can be completely separate as well. Tools able to satisfy this need of document sharing can include Google Drive[14], external wikies, Blackrock iShares[15] or Dropbox[16]. These repositories obviously contain information relevant to the project describing its progress and intermediate and partial outputs.

### 2.3.4 Communication Tools

Communication is an essential activity in collaborative projects. This fact is demonstrated e.g., by it being one of the central points of the Agile Manifesto. Broad, open and effective communication channels have to be established between different stakeholders of a project. Customer needs to share its business case and comment on the current state of the work, management needs to get status reports from development team and the team has to internally communicate on the day to day activities to perform them effectively.

Furthermore, it is preferable for all communication to be stored as human memory is unreliable and the recollection on what was decided a week ago can differ vastly among the participants. Of course face-to-face communication can (and if possible should) play a part. But as it cannot encompass everything and its harder to document (e.g., in audio or video recordings), the bulk of the project communication happens elsewhere. To support this,

[14]https://www.google.com/drive/
[15]https://www.blackrock.com/corporate/ishares-global
[16]https://www.dropbox.com/

project staff employs different communication tools. Some of the communication capabilities can be included in previously mentioned tools. Document sharing, meeting notes and progress tracking are means of conveying information as well.

Some tools even come with instant messaging capabilities built-in. Nevertheless, external tools are also an option. The most common means of communication is email. The OSS projects even make their email archives publicly accessible for documentation and analysis purposes. Other communication applications can include Slack[17], Skype[18], Facebook Messenger[19] or other instant messaging and teleconference tools. Records of communication can provide deeper insights into the inner working of a project and the origins of ideas before they are captured in purpose-specific tools (e.g., for change management).

### 2.3.5 Others

Other tools falling into the definition of ALM can include activity- or phase-specific tools. These can include requirements management tools (e.g., IBM Rational Dynamic Object Oriented Requirements System)[20], design tools (e.g., Sparx Enterprise Architect[21]), quality management tools (e.g., IBM Rational Quality Manager[22]), or deployment tools (e.g., Elastic Cloud[23]).

Though these can also hold valuable information about the project, they are at least for now out of scope of our work, which focuses mainly on tool categories specified in earlier subsections. Our rationale is that issue-tracking tools are capable of capturing requirement and testing tasks in form of tickets, an option a sufficiently large number of projects (especially OSS) takes advantage of, and resulting artifacts from these, along with analysis and design activities are stored in knowledge base or VCS tools. That gives us at minimum the knowledge of about their existence, authors, creation and modification dates and several other attributes which are currently sufficient for our purposes. The particular format, source tool and details about content can therefore be omitted.

[17]https://www.slack.com/
[18]https://www.skype.com/
[19]https://www.messenger.com/
[20]https://jazz.net/products/rational-doors-next-generation/
[21]https://www.sparxsystems.eu/start/home/
[22]https://jazz.net/products/rational-quality-manager/
[23]https://www.elastic.co/cloud

### 2.3.6 Full-fledged ALM Tools

We consider a software to be a full-fledged ALM tool if it is able to really manage and track the whole lifecycle of a project from business case, requirements and design, through implementation, testing and deployment, to even maintenance, operations and retirement (if such phases are a part of the project). However, due to their usually modular and plug-in-based or integration supporting architectures many singular tools posses such capabilities. This is logical as from the business perspective of their developers it is easier to market and sell specific customizable solutions with capable of mutual integration than huge "all-powerful" systems. Therefore we use the term full-fledged ALM tool for either a complex suites of tools, like GitLab[24], Atlassian Jira[25] and IBM Rational solution for Collaborative Lifecycle Management (CLM)[26], or tools from specific category with explicit integration options that complete their ALM capabilities (e.g., Redmine and Assembla).

## 2.4 Similar Work

This section describes related work already done in the realm of software process patterns and anti-patterns with the description of features distinguishing our work from each individual instance.

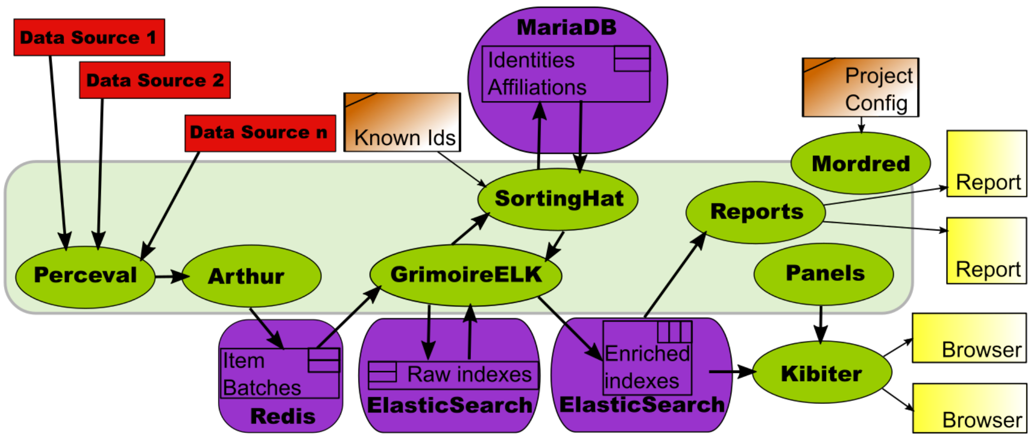


Figure 2.7: GrimoireLab architecture [24]

[24]https://gitlab.com/
[25]https://www.atlassian.com/software/jira
[26]https://www.ibm.com/us-en/marketplace/application-lifecycle-management

### 2.4.1 CHAOSS

Community Health Analytics Open Source Software project (CHAOSS) [119] is a complex system of tools, metrics and methodologies aimed at OSS. Its goals are to measure health and sustainability of OSS projects to support informed decisions for contributors of those projects (e.g., where to put their effort and see the impact the are making), the OSS communities (e.g., attracting contributors, reward the valuable ones and ensure product quality), and companies (e.g., which communities and project to engage with and evaluate employees effort within OSS projects). The CHAOSS community has a substantial social media presence, organizes events (such as CHAOSScon) and is sponsored by The Linux Foundation.

One of the projects of the CHAOSS community is GrimoireLab [24] and their tools for software analytics. The purpose of this tool set is to gather, analyze and visualize data from various ALM tools dedicated to OSS development contributing. Further use of this data and visualizations is vast and left to the users and there is no specific purpose of the research on top of collecting and presenting the data. The overall architecture is displayed on Figure 2.7.

> *"In the figure above, GrimoireLab components are represented in the pale green box. Bold arrows show the main data flow: from data sources to Perceval (which retrieves them), to Arthur (which schedules retrieval batches and stores results in Reddis), to GrimoireELK (which stores retrieved items as raw indexes, and then uses them to produce enriched indexes, both in ElasticSearch), to Reports (to produce specialized reports) or Kibiter (to visualize in actionable dashboards).*
>
> *GrimoireELK uses SortingHat to store all the identities it founds in a MariaDB database. SortingHat uses lists of known identifiers (usually maintained in configuration files) and heuristics to merge identities corresponding to the same person, and related information (such as affiliation).*
>
> *All the process is configured and orchestrated by Mordred, which uses its own configuration about, for example, which data sources to use."*[24]

Perceval, a collection of data pumps for various ALM tools, puts out data in a form of JavaScript Object Notation (JSON) files for further processing. It can then be analyzed and presented in dashboard-based tool called Kibana. Kibana is a browser-based analytics and search dashboard plug-in for Elasticsearch. GrimoireLab uses its own fork of the Kibana software, a browser-based analytics and search dashboard plug-in for Elasticsearch, called Kitbiter. The resulting and customizable dashboards can be seen on Figure 2.8. Kibana and as well as all the GrimoireLab tools and CHAOSS projects are OSS themselves.

As a point of difference to our work, the approach is to analyze data from one project representation from one tool at a time. Meaning one cannot combine data about one project from various source and therefore would have to rely on only those projects that utilize full-fledged ALM tools eliminating those that use e.g., Bugzilla, SVN and external wiki. The other option is to make judgment calls based solely on the incomplete data from one tool. Moreover, the data are tool specific and a unified format across tools is not used which makes comparison or analysis over several projects from different sources impossible.

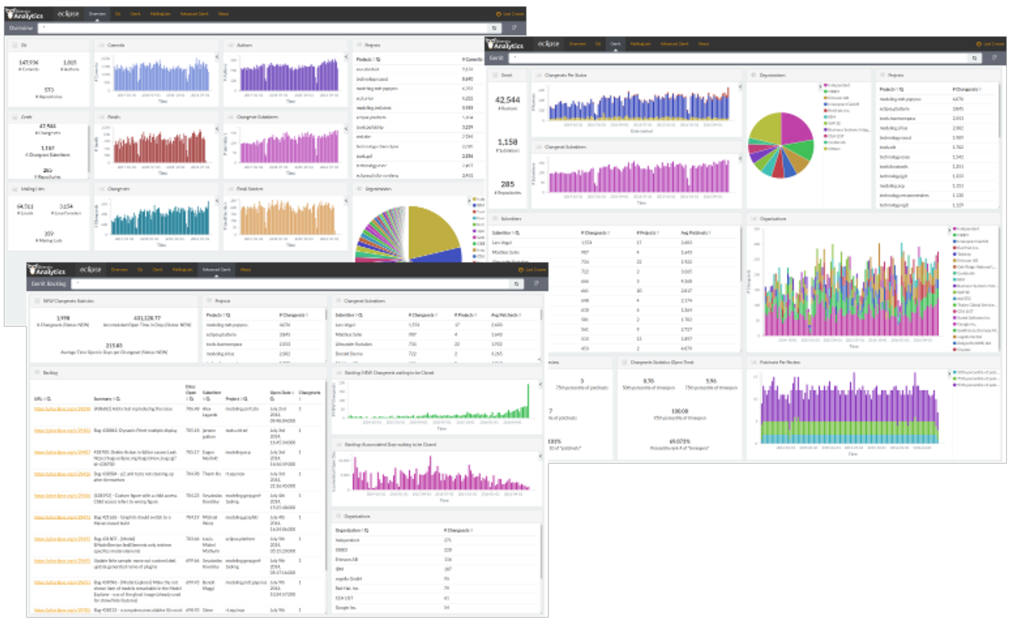


Figure 2.8: GrimoireLab's Kitbiter (Kibana) dashboards [24]

### 2.4.2 Bloof

Draheim and Pekacki are developing an approach to analytically process project data [31]. In their efforts they mine data form source code repositories and analyze it from a process-centric viewpoint using metric on collaboration, productivity, project evolution, etc. Their Java-based experimental tool called Bloof utilizes data model (see Figure 2.9) capturing realities of a analyzed project such as developer, file and revision with expected attributes.

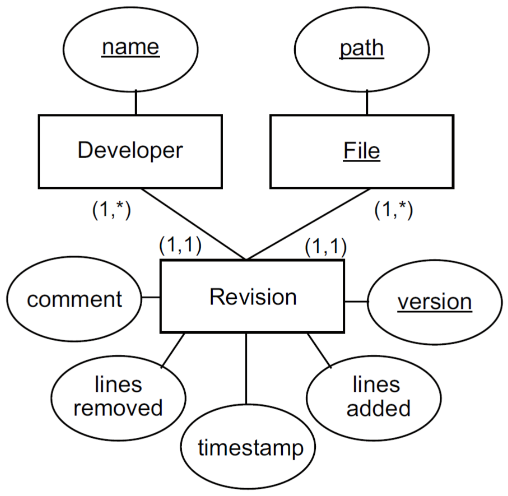


Figure 2.9: Data model of Bloof [31]

Bloof uses an Extract-Transform-Load (ETL) layer (extraction, transformation, loading) to collect and store data in a database and provides an analytical processing interface for performing data queries and allowing for user defined custom queries. The queries are realized in two ways, standard Structured Query Language (SQL) format and predefined analytical compound queries, that can be parameterized for complex analysis problems. The approach uses process-centric analysis focusing on developers activity throughout the project with metrics like lines of code (LOC) changed per day, collaborative vs. total changes, distribution of changes between modules, average time for file change, etc. Bloof includes graphical user interface (GUI), Bloof Browser, which displays the charts and tables to visualize the metrics calculated.

Though a valid starting point, the approach misses an opportunity for wider context focusing solely on data form VCS and not incorporating other readily available project data from sources like issue-tracking tools and mailing lists.

### 2.4.3 SoftChange

SoftChange is an OSS tool for extraction and validation of project data described by German and Mockus in [41]. Through this tool they are striving to create and test theories about the nature of the OSS projects as well as compare them. SoftChange is a collection of scripts and analyses data from mailing lists, CVS logs, ChangeLog files, and defect tracing databases, specifically Bugzilla. It also identifies all contributors to software change using cross-links records.

Though it collects and analyzes data from sources of different kinds (VCS, issue-tracking, mailing lists), it heavily focuses on specific toolset (CVS, Bugzilla) limiting the potential dataset to only the projects making use of these specific tools and no others.

### 2.4.4 Codeface and PaStA

Research efforts by Joblin, Mauerer, et al. to analyze project's community structures, interactions among contributors, and their evolution in OSS projects resulted in an experimental tool called Codeface. Building on the work of e.g., Jermankovics et al. [57], and Zimmerman and Weissgerber [124], the approach is based on analyzing data from code repositories (mainly Git) and mailing lists with regards to direct and indirect communication between developers. Codeface utilizes both Application Programming Interface (API) and web-crawling to gather data, which it stores in a database. The researchers use a fine grained function-level approach [61] to establishing interaction between developers working on the same source code artifact, which is the approach that separates their efforts from previous and other concurrent research groups using file-level analysis only. They also use the sliding window method to investigate the evolution of the project's community [60]. They then construct graphs of the overall community, calculating a set of metrics with focus on different aspects. These include:

- **Involvement** – The distinction between core and peripheral contributors which Codeface calculates based on network-based operationalization rather than count-based one [59] (another aspect distinguishing the approach from others).
- **Turnover** – The frequency of core developers becoming peripheral and vice versa, or leaving the project completely.
- **Scale-freeness** – A property of developer which has characteristics including robustness to perturbations. It means that a removal of a random node from the network will likely not damage the connectivity of the network
- **Modularity** – A measure of division of the developer network into loosely connected clusters of heavily interconnected nodes.
- **Hierarchy** – The phenomenon of small cohesive groups are embedded within larger and less cohesive groups. Hierarchy indicates a centralized command-and-control structure.

Their findings suggest that while starting as such, the organizational principles of communities involved in OSS are non-random, especially with increasing size of the staff. The current focus of the Codeface project is to also incorporate data from other sources like issue-tracking tools to enhance the completeness and accuracy of their results. They also intend to investigate the relationship between the community attributes a metric values and quality of the product produced by the projects.

The focus on the community graphs and developer network aspect of projects means that most of the approach cannot be utilized for our purposes. The data model includes domain specific entities and attributes obsolete for our intended usage and lack other we would need. The analytical layer is even more distant to our focus. However, Codeface and our approach can prove complimentary and can benefit from each others ideas and results.

Another tool resulting from related research by Ramsauer, Mauerer et al. is called Patch Stack Analysis (PaStA) [98]. This tool looks at data about commits in VCS (again primarily Git) and their code increments (patches) and their transfer from being suggested in the mailing list communication records and actual appearance in the repository. This is done with detection of similar patches, through which PaStA is also capable to recognize the occurrences of the same patch in patch-stacks and main development branch giving researches the insight into the history of the patch and its way from patch-stack to mainline or the other way. It can also detect splitting of a patch into two (or more) and the compression of multiple patches into one. These attributes distinguish the research from similar ones, like Canfora et al. [22]. The main goal of the work is to estimate the maintenance costs in software development. The future work includes patch classification by the purpose of the introduced patch (e.g., bugfix, new feature, licensing, refactoring, etc.). For this purpose the researchers are conducting a survey on the patch classes which people in OSS recognize, and they also utilize a machine learning algorithms for automatic classification of the patches. In near future PaStA is scheduled to be integrated into Codeface to enhance its analytical abilities and provide deeper insights.

Both tools are developed as OSS themselves mainly in Osterbayerische Technische Hochschule (OTH) Regensburg and are implemented with technologies such as Python, R and SQL. Their proprietary forks are currently used by both research (University of Passau) and industry (Siemens) communities for different purposes.

### 2.4.5 CoSEEEK

The research efforts of Grambow, Oberhauser and Reichert center on SPI through adjusting the workflow of individual developers. They call the approach and an experimental tool based on it Context-aware Software Engineering Environment Event-driven frameworK (CoSEEEK) [85]. They also proposed a specialized Software Engineering Workflow Language (SEWL)

[46] represent executable workflow models. CoSEEEK uses technologies such as AristaFlow, SPARQL, Hackystat, Drools, SensorBase and others. Figure 2.10 shows the overall concept of the CoSEEEK architecture.

The tool is connected to a developers environment through agents in various SE tools the developers uses in his work. This includes virtually all software a developer uses while working on a project from VCS and issue-tracking systems, to email client applications, to multiple Integrated Development Environments. The agents actively wait and react to events (i.e., actions taken by the developer in the tool) that change the artifacts (e.g., source code) or the tools themselves and their data. Events are extracted and processed by designated modules [45] and captured in data storage while the rules processing module analyzes tool data.

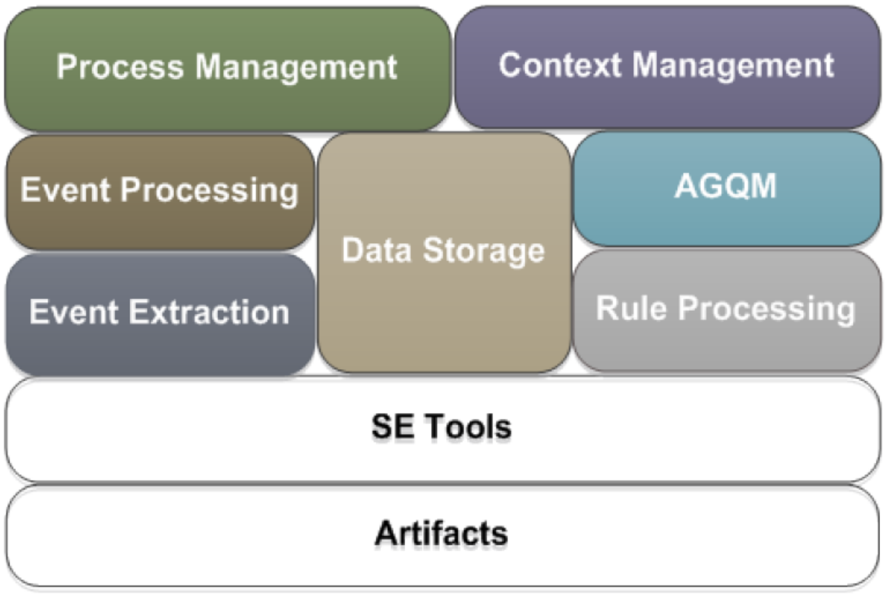


Figure 2.10: Conceptual architecture of CoSEEEK [44]

The data is then filtered by the Agile GQM (AGQM) module to gauge the project quality according to its goals and to propose appropriate quality measures. The process management module compares the previously defined workflows with the reality of the project to identify possible injection points for the proposed steps and the context management module collects high-level information of all project areas (e.g., developer skills) to make the contextually aware decisions on the measure injections [44].

This way, the personal workflow of an individual is reconstructed, and then used to for example identify possible time slots which could be utilized for performing quality measures if necessary (based on project quality state and goals). If contextually appropriate, the measures are injected into the work-flow and the user is notified. The end goal of this is to incentivize developers to address quality risks, through performing e.g., testing activities, as soon as possible which presumably should lead to increase in the overall quality of the software product.

The main goal of CoSEEEK research differs from ours. The main point of difference is the need of CoSEEEK approach for nearly immediate collection of data changes from the source tools and reactions to them. The events are captured as they occur and the results are delivered as soon as possible to maximize the impact of proposed changes to the workflow. Our approach is more aimed at checking the progress of the whole team at discrete moments in time on e.g., periodic reviews or even post-mortem analysis. The process patterns take days, weeks or even longer to manifest in detectable fashion and require project- and team-wide analysis of the data for which the usage of agents and event-driven recalculations might prove unnecessarily burdensome in respect to computational power requirements, if possible at all.

### 2.4.6 YOSHI

The work of Tamburri et al. [116] presents Yielding Open-Source Health Information (YOSHI) tool focused on organizational aspects of OSS communities in OSS development. They provide an automated support to evaluate social and organizational characteristics mapping communities to known organizational and social structures and providing insights vital for individual and organizational decision making. Opposite to e.g., Codeface, YOSHI does not focus on a community on a single project, but investigates even involvement of a particular developer in different projects and communities. For these purposes it analyzes primarily data from Github and measures community characteristics such as:

- **structure** – based on the interactions between people inside an observed group and their frequency,
- **formality** – average membership type (in GitHub contributor or collaborator) divided by the milestones per project-lifetime ratio,
- **engagement** – levels across the community are established as a the member averages of the measurements like commit/pull request comment total and frequency, number of repository active members, watchers, subscriptions, distribution of commits and collaboration on files between users,
- **cohesion** – based on number of followed and following members of the same community with shared expertise overlap,

- **longevity** – difference between first and last commit for each member of the community,
- **geodispersion** – the degree of distribution of the community member around the globe.

Based on these measurements the researchers recognize nine types of organizations/communities [115]. This gives the researchers an opportunity to investigate community health, reuse of type-specific best practices, diagnosis of organizational ant-patterns, four of which are explicitly specified in their work (*Organisational Silo*, *Lone Wolf*, *Black Cloud* and *Bottleneck*). To detect these anti-patterns (or community smells) they make use community metrics provided by the Codeface tool (see Section 2.4.4).

The data from for analysis is collected from GitHub using a Java API. The overall architecture of the tool is captured in Figure 2.11 Same as a number of the previously mentioned research tools, YOSHI implementation is made public as an OSS as well.

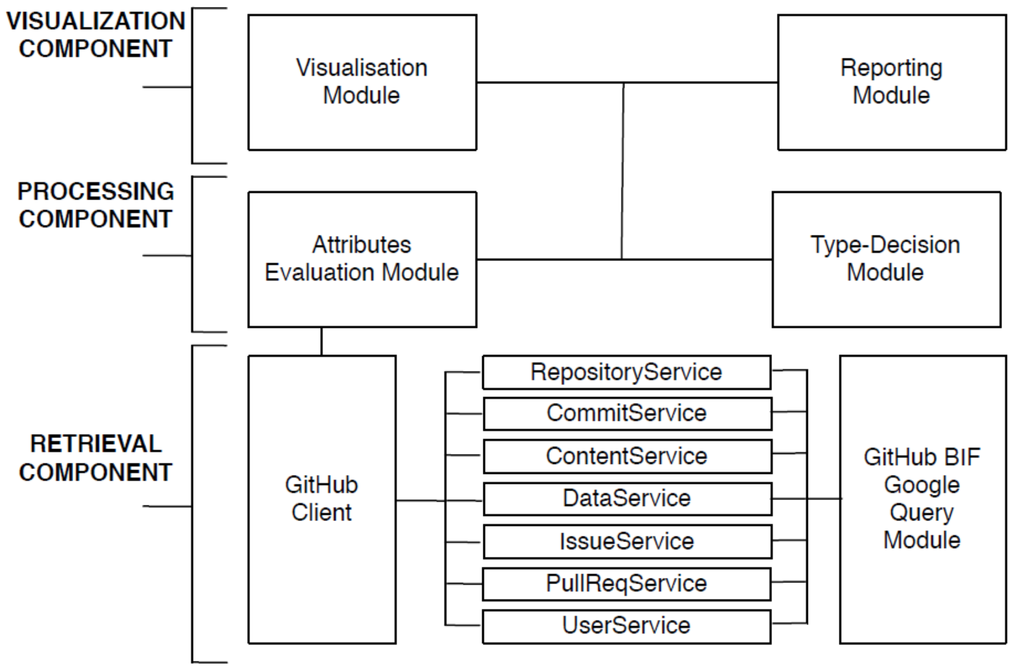


Figure 2.11: YOSHI high-level architecture [116]

Because of its focus on communities, people data across different projects and GitHub data only, YOSHI is not directly compatible with and therefore not usable for our purposes. However, the utilization of Java API is a straightforward way of obtaining data and when computed over a single project data only, we may be able to use Codeface metrics to detect anti-patterns specified in [116].

### 2.4.7 Vranić et al.

Researchers in Bratislava, Slovakia focus their research on organizational patterns. Specifically they are interested in practices of agile and lean methodologies and their adoption in individual organizations and projects. To this end they developed a method of animating organizational patterns as text adventure games [38]. The space of scenarios of the game is expressed via UML state machine diagrams. The game presents the user with a scenario, which is hypothetical, but based on real-world project situations and experiences, and several options on actions to take, taking them into a new state with new options, and so on. The purpose is to educate the user on particular agile and lean practices and their usage through this soft simulation and experience working from the hypothesis, that the approach has a better chance on successfully demonstrate the benefits of the practices than a pure theoretical textual description.

The work deals with hypothetical situations and aims to educate and train users through them. It does not deal with data collection and pattern detection. But, our two approaches could be combined in scenarios in which an anti-pattern is detected, team members then trained to adequately respond to it, and the efficiency of the training judged by the subsequent measurement still detecting the anti-pattern, or discovering its disappearance.

### 2.4.8 Hipikat

Čubranić and Murphy at University of British Columbia came up with a method to identify relevant and important artifacts in OSS projects [26]. The purpose is, among other uses, to get new contributors coming into a project up to speed by providing them sources of with the highest concentration of knowledge of use to them. This is important because in OSS projects it is not always the case that another team member can function as a mentor or trainer for newcomers. Such a person with both a sufficient overview of the whole project and information sufficiently detailed for the specific role or interests for the newcomer may not be available or even exist in the project. The experimental tool they use is called Hipikat and analyzes data from CVS, Bugzilla, mailing lists and online documentation.

The domain model of Hipikat for storing data is shown in Figure 2.12. The tool is a client-server application. The server is a web application running on Apache Tomcat and the client is implemented as an Eclipse plug-in, they both communicated using Simple Object Access Protocol (SOAP). The data gathering from CVS is done through parsing history records obtained by command line commands, from Bugzilla and online documentation via web page parsing (web crawling). This is done by monitoring, i.e., automatically periodically invoking the commands and parsing the sources. By comparing various attributes of artifacts Hipikat then infers the relationships between them. User then specifies an artifact of interest and Hipikat follows the relations to other artifacts to be recommended to the user. The architecture and flow is captured in Figure 2.13.

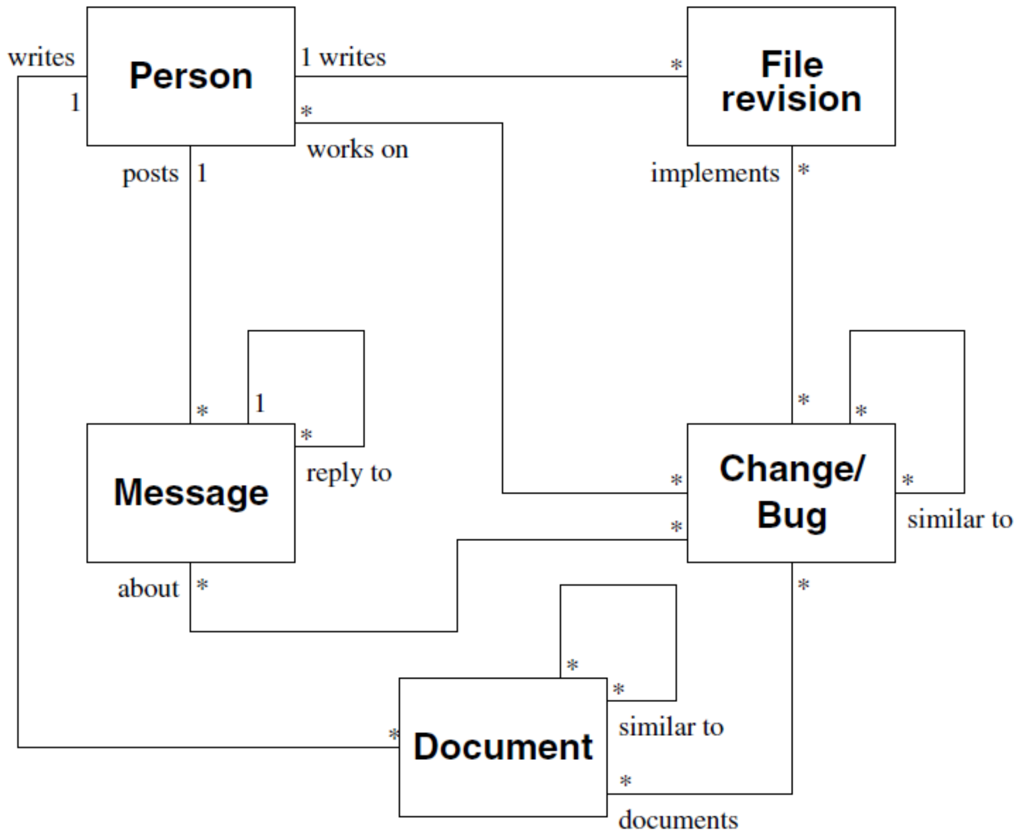


Figure 2.12: Domain model of Hipikat [26]

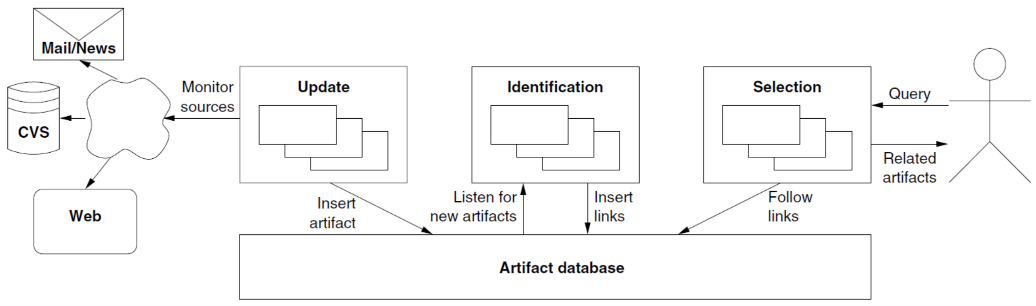


Figure 2.13: Hipikat architecture [26]

The source tools Hipikat gathers data from puts it on equal footing with SoftChange (see Section 2.4.3) in terms of data sources. They are varied in terms of types, but needlessly specific in that it uses only one representative of each type.

### 2.4.9 SPARSE and PROMAISE

Settas, Stamelos et al. created two frameworks for capturing the knowledge on software management anti-patterns and its effective communication among SE practitioners. The approaches are Symptom-based Antipattern Retrieval Knowledge based System Using Semantic Web Technologies (SPARSE) [107] and The Software Project Management Antipattern Intelligent System (PROMAISE) [105]. They base their approaches on Web Ontology Language (OWL) for capturing the anti-patterns and their symptoms, and Bayesian (Belief) Network (BN) for modeling the probabilistic relationship among a set of variables. In their work they aim specifically at PM anti-patterns based on the heterogeneity of personalities and character traits of developers [104, 106]. These can result in ineffective communication and lack of motivation to collaborate and therefore lead to problems and uncertainty in managing the project.

This work is aimed at completely different set of (anti-)patterns, as personality traits and temperaments can hardly be captured in the project data from ALM tools and are therefore out of scope of our research.

## 2.5 Software Process Modeling

This section focuses on established software development process capturing approaches, languages and models with the potential to store the gathered project data from ALM tools in.

A large number of such entities have been developed and proposed over the years and so this work will focus only on several most well-known and potentially relevant example. The whole field of software process modeling languages (at least up until 2011) has been reviewed and summarized by García-Borgoñón [40]. As is visible in Figure 2.14, the trend in the last decade or so has moved towards using metamodels (in bold frames), therefore these are primary focus of this thesis.

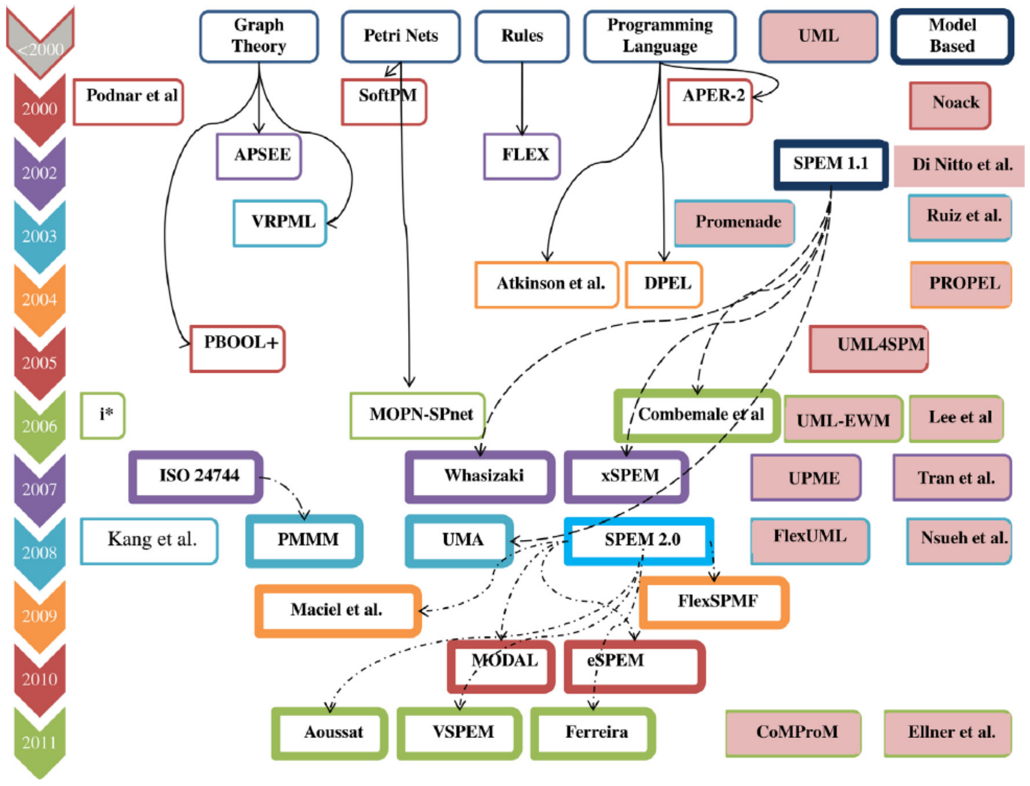


Figure 2.14: Software process modeling languages overview [40]

## 2.5.1 SPEM

Software & Systems Process Engineering Meta-Model (SPEM) is probably the most well-known mean of modeling software development process. SPEM was original established in 2002 in version 1.1, SPEM 2.0 [86] has been introduced in 2008.

SPEM does capture the base theoretical concepts of software development process such as roles, artifacts, activities, etc. and goes into great amount of detail from there. It separates the definition of the concept and its instantiated usage in the process, also known as a descriptor. It therefore provides the user of defining the building blocks first, and subsequently create multiple descriptors for each of them to be used as a building block of the overall process model.

An IBM tool Rational Method Composer (RMC) is a software application implementing SPEM modeling, allowing its user to create reusable process models and their components as well as tailoring abilities to slightly alter a defined process based on a context of a specific software project.

SPEM has also spawned several purpose-specific metamodels derived from it. A few of those potentially interesting and relevant for our future work we will now describe.

#### eSPEM

The enactable SPEM (eSPEM) [32] was developed as an extension to SPEM 2.0 in 2010. It was developed to allow for computer-aided enactment of a process model by providing an extension to SPEM based on UML metamodel focused on fine-grained modeling generally considered out of scope of SPEM itself.

#### vSPEM

Also an extension of SPEM 2.0, variability SPEM (vSPEM) [76] was introduced in 2011. vSPEM is a concise notation specific to the process domain. It addresses limitations of the SPEM 2.0 variability mechanisms. It allows for identification of variability points in the process, each of which is replaced by exactly one variant (one of the beforehand specified options) during the process instantiation.

### 2.5.2 OSLC

Open Services for Lifecycle Collaboration (OSLC) [89] is an effort to provide a standardized baseline for ALM tools in terms of the realities they should capture along with their structure and naming conventions. The purpose of this is to foster an easy way to migrate data from one tool to another with similar purpose (e.g., change management). OSLC provides structure for data in all standard activities in software project, such as requirements management, PM, quality management, configuration and change management, etc. Each of its parts is subject to independent versioning. The most important specifications of OSLC for our work at this point are Change (currently in stable versions 2.0; 3.0 as a draft) and Configuration Management (currently as a draft only).

### 2.5.3 ISO 24744

ISO/IEC 24744 [102] standard provides the Software Engineering Metamodel for Development Methodologies (SEMDM). It serves as a source of an ontology for standards harmonization. The metamodel is a semi-formal language mainly focused on the ability to describe software methodologies, though it has been pointed out that it contains nothing that would hinder its use beyond this specific domain. Though similarly focused, SEMDM goes into more detail than SPEM especially in the area of work products (artifacts), producers (actors), stages (temporal aspects and relations) and model units. Ruy et al. performed and ontological analysis of SEMDM to identify several problems in the metamodel.

#### PMMM

An ISO 24744 derivative, Process Meta Meta Model (PMMM) [55] provides a method and language to model domain specific process management metamodels. It is therefore, and as the name suggests, at least a level above even metamodels. It contains three basic entity types: Nodes, NodeAttachments and Flows. It is a language to specify lower-level domain-specific languages and is therefore out of scope of this work, though some inspiration and concepts may be interesting for potential adoption.

### 2.5.4 ISO 29110

ISO/IEC 29110 [72] is a standard dealing specifically with very small entities having up to 25 people. This means that, while useful in particular instances, it is not suitable for majority of our work which is not constrained by any upper bound in terms of team/organization scope. Nevertheless some concepts can be adopted, especially when analyzing these very small entities, or extrapolated.

### 2.5.5 BPMN

Business Process Model and Notation (BPMN) [87] is a well-known technique of modeling process models. It provides a notation to comprehensibly describe business processes in a graphical, diagram format. It has the capability to capture elements like events, tasks, transactions, gateways, message flows, and artifacts. Current version 2.0 released in 2011 among other features added displaying different perspectives and Extensible Markup Language (XML) schemata for model transformation. BPMN is obviously focused on business activities and provides overview and insights to support executive decisions.

## 2.6 ALM Tools Data Mining

A substantial amount of effort has been done in the realm of mining data from ALM tools. This section serves as an overview of the literature gathered on this topic not mentioned in the previous section.

### 2.6.1 Mining Software Repositories

Hemmati et al. [50] performed an extensive review of 117 papers on MSR published between 2004 and 2012. They extracted 268 comments from these papers and identified four high-level themes using grounded theory methodology. These themes are:

- data acquisition and preparation,
- synthesis,
- analysis,
- and sharing/replication.

The researchers also identified several recommendations in each of these four themes. Bird et al. [14] focused their research of comparing benefits and pitfalls of mining Git in contrast with centralized VCS (such as Apache Subversion), while Kalliamvakou et al. [63] did a similar work on mining GitHub. Jensen and Scacchi [56] explored techniques for discovering processes from OSS repositories employing artificial intelligence technology. The work of Fisher et al. [35] deals with combining data from VCS and bug-tracking systems to populate and investigate a software release history database representing the evolution of a software project. The Reengineering of Software Evolution (ROSE) [125] tool developed by Zimmerman et al. strives to predict further changes in software based on mining version histories and investigating the likelihood of a set of programmers, who changed one function, to also change another. Herzig and Zeller [53] provide a guide to mining bug data and use the resulting knowledge to model, estimate and predict source code quality. Mockus and Votta [81] mine the textual description of changes to understand the maintenance activity and the relationship between type and size of a change and the time needed to carry it out.

### 2.6.2 Cumulative sources

Several projects have been launched to gather and store huge amount of project data. These can be used as a substitute for directly mining the ALM tools repositories and lead to faster results because of the concentration of data and their relatively similar format in these sources and a unified access due to the available API. One such source is the Software Heritage project [30] with its goal to collect, preserve and share project data that could otherwise be lost due to e.g., decommissioning of the original data sources, such as Source Forge. The ever-growing storage of Software Heritage[27] currently contains more than 8 million projects, one billion commits and 4.5 billions source files.

Other source of this nature is the dataset of GitHub projects made available by GitHub itself specifically for research purposes [43]. The dataset is is in the form of a data warehouse and analytics platform called Big Query and it is equipped with web user interface (UI), command.line tool and representational state transfer (REST) API with client libraries in Java, .NET, or Python. The dataset is currently over 4 TB in size.

## 2.7 Pattern Representation and Detection

The general concept of a pattern was defined in Section 2.1.5. This section provides an overview of the research done in the field on PM pattern and anti-pattern detection and representation.

### 2.7.1 Textual Descriptions

Through (anti-)pattern sources in literature are plentiful, structures for textual description of (anti-)patterns vary heavily. Here we provide some example of the (anti-)pattern description structures, mainly taken from [17] and [71].

#### Degenerate Form

A description in textual form without any structure, template or separate content sections for various aspects of the pattern. It is up to the reader to identify by the free text given by the author, if the text describes a pattern at all, its context, symptoms, consequences, solution and other properties.

#### Pattern Templates

**Alexandrian Form** captures name, problem and a solution lead by the word “THEREFORE”. It partitions the motivation and the actions.

[27]https://www.softwareheritage.org/

Conventionally also includes a diagram.

**Micro-Pattern Template** (also called Minimal) consist only of name, problem and solution in short description.

**Mini-Pattern Template** decomposes the problem to context and forces and adds benefits and consequences to the solution to focus on the teaching elements of the pattern. It can be found in two forms: inductive, focusing on the pattern applicability, and deductive, focusing on the outcomes of the solution.

**Formal Templates** add many other sections to the descriptions, like relations to other patterns, aliases, intent, motivation, known uses, variants, implementation, examples, etc. The particular set of sections depends on the domain and the pattern itself. The goal of the formal templates is to describe the patterns in as much detail as necessary for them to be understood, correctly identified and applied by the users. Examples include Gamma et al. (a.k.a. Gang-of-Four) Template for micro-architecture-level patterns [39], or System of Patterns Template for idioms, application and system level patterns [20].

**COBRA Design Pattern** is a pattern description form devised by the authors of [17]. Its main purpose is quick comprehension and limited amount of long free-text descriptions. Therefore, it prioritizes the most important sections first, like keywords, intent and solution diagram, and pushes the longer sections further down. Furthermore, it uses a common reference model for frequently repeating aspects like context and forces, which are captured in the pattern itself in the form of keywords and described elsewhere.

### Anti-pattern Templates

**Pseudo-AntiPattern Template** is a short degenerate form of anti-pattern, often in disparaging terms, subjective and because of the lack of structure cannot be used as precise definition and analyzed.

**Mini-AntiPattern** is similar to the Mini-Pattern Template in that it consist of name, problem and solution. Only, being aimed at anti-patterns in a sense of frequently applied bad solutions of common problems, the problem is actually the bad solution and then the better (refactored) solution follows.

**Full AntiPattern Template** as used by Brown et al. [17]. It consists of the following sections:

- Name,

- Also Known As,
- Most Frequent Scale,
- Refactored Solution Name and Type,
- Root Causes,
- Unbalanced Forces,
- Anecdotal Evidence (optional),
- Background (optional),
- General Form of this AntiPattern,
- Symptoms and Consequences,
- Typical Causes,
- Known Exceptions,
- Refactored Solutions,
- Variations (optional),
- Example,
- Related Solutions,
- and Applicability to Other Viewpoints and Scales.

**Laplante-Colin Structure** is a template used by Laplante and Neill [71]. It is less formal than the previous form by Brown et al., and focuses more on identification of the dysfunctional situation and remedies for all involved. The template is structured as follows:

- Name,
- Central Concept,
- Dysfunction,
- Vignette,
- Explanation,
- Band Aid,
- Self-Repair,
- Refactoring,
- Observations,
- General Form of this AntiPattern,
- and Identification.

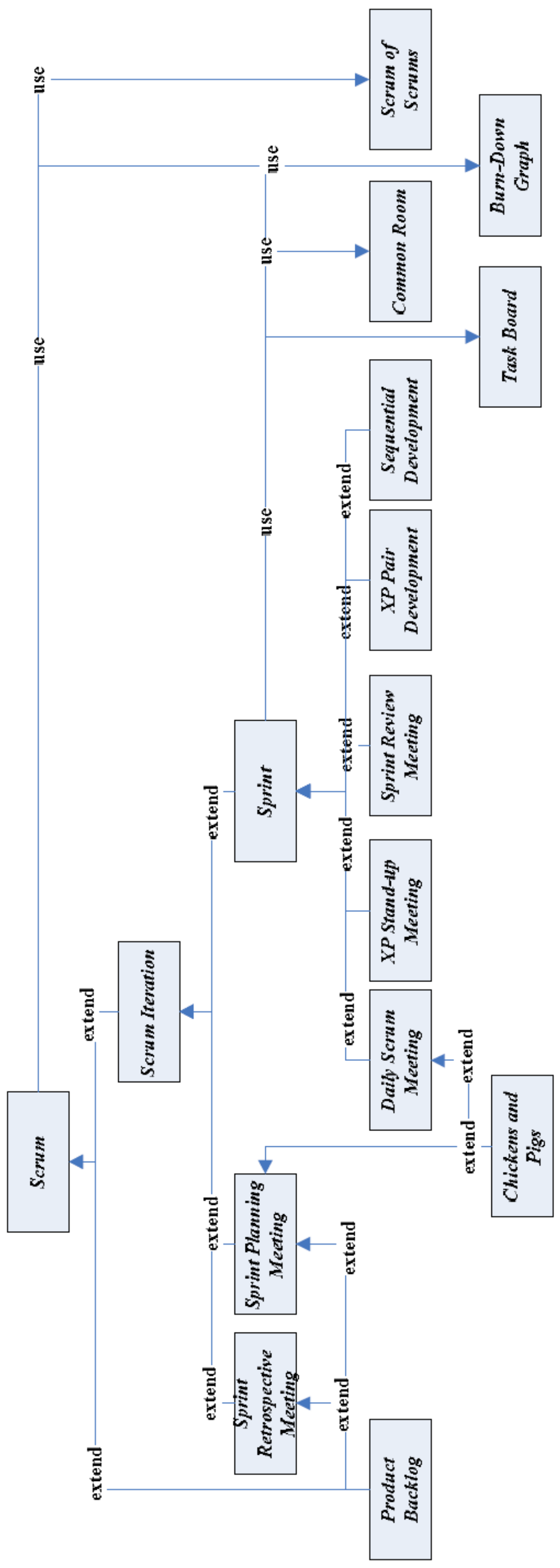


Figure 2.15: A Process Pattern Language for agile methods [79]

### 2.7.2 Languages and Ontologies

Ontologies are another technique for used for pattern representation, usually working with a proprietary language for easier automated machine processing than pure textual description in natural language. As mentioned in Section 2.4.9, SPARSE and PROMAISE use web-based ontologies to detect anti-patterns in a knowledge base based on symptoms selected by a user. CoSEEEK (see Section 2.4.5) also uses web ontology, SEWL, for example to automate process assessment based on Capability Maturity Model Integration (CMMI) and other standards [47].

Research team [79] proposes a Pattern Process Language (PPL), and specifically its version for agile methodologies – Agile PPL (APPL). Figure 2.15 shows APPL. In their work, language means the set of patterns and their relationships to each other. Therefore the language cannot be general for the whole domain of software development processes, since the set of practices (i.e., patterns) each methodology or process model composes of is different in some measure. That is why an agile specific PPL had to be constructed.

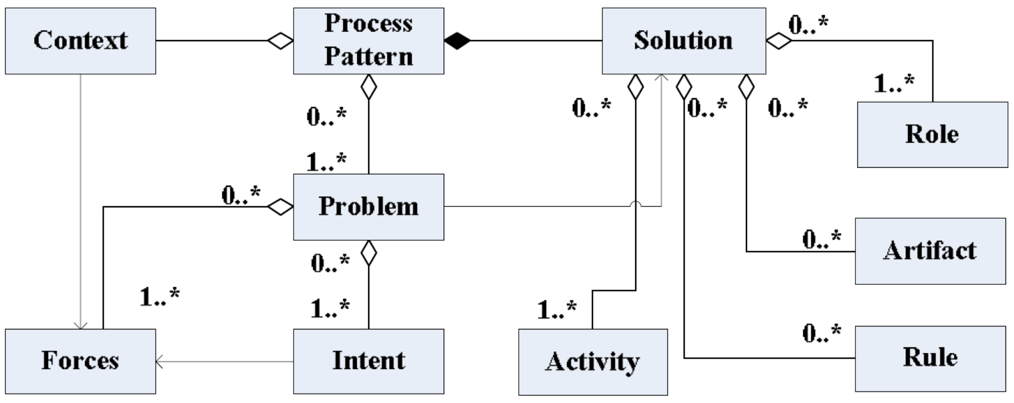


Figure 2.16: Process Pattern Language primary elements relationships [79]

A language in this sense is basically an interconnected set of practices, which can be customized by adding, removing and modifying patterns. The patterns themselves are captured in a schema of primary and accessory elements and their relationships. Figure 2.16 show the relationships between primary elements.

In [117] researchers propose and formally define A Process-model Query Language (APQL), a language for querying business process models independent of their notation based on semantic relationships between tasks. Avoiding committing themselves to specific keywords or to the order of statements, the researchers opted to use an abstract syntax.

Research being conducted by Roa et al. [100] focuses on detection of onthology-based anti-patterns in BPMN models for the purposes of verification of the models' behavior.

### 2.7.3 Models

Modeling through graphical notation (i.e., diagrams) is another way of capturing process patterns. Because process can be viewed as either a large complex pattern itself or a composite of smaller patterns, the process metamodels described in Section 2.5 can also be used for modeling individual patterns. In an effort to compare model notations used in practice predominantly for business process modeling, authors of [120] identified seven problems related to the process modeling.

Models often rely and build on languages and ontologies (see Section 2.7.2). The output of producing a diagram can be not only its image for better human comprehension, but also a description of the model in the particular language or ontology usually using technology like XML or JSON. This format is then eligible for machine processing, an approach employed by e.g. [23] on SPEM models.

Researchers in China [48] developed a predicate-based Control-flow Anti-Pattern Description Language (CAPDL) to define and detect control-flow anti-patterns in process models, specifically in BPMN. They decompose anti-patterns they want to detect into rules and further into predicates in a fashion not dissimilar to BN or Goal-Question-Metric (GQM) approaches.

Furthermore, Ramsin et al. identified patterns for Component Based Software Engineering (CBSE) [67] and Aspect-Oriented Software Development (AOSD) [64] by studying established methodologies, processes and practices in the respective domains. They described these patterns using a special model notation of their own device.

For process (and therefore pattern) models, using of UML activity diagrams could also be considered, though it is not well-suited for ALM data specifically since the activity, input, output and actor elements used by the notation lack the means of detailed specification of all their attributes present in project data. Similarly, process algebras could be utilized for modeling process patterns but are insufficient to model e.g., pattern representing the particular form and content of an artifact.

### 2.7.4 Bayesian Networks

A frequent technique for process problem or pattern modeling and detection is the usage of Bayesian (Belief) Networks (BNs). BNs are directed, acyclic graphs representing probability distribution. Examples like [107] and [105] were already mentioned (see Section 2.4.9), but others exist.

Khomh et al. presented Bayesian Detection Expert (BDTEX) [65], a GQM based approach to build BNs from the definitions of coding anti-patterns and discussed advantages of BNs over rule-based models. In their work, symptoms specifying anti-patterns are selected by a quality analyst, and calibration is done automatically using Bayes' theorem. They claim two main benefits to their approach as compared to previous approaches. BNs work with missing data and can be tuned using knowledge of quality analysts, and candidate classes (i.e., potential anti-patterns) are associated with probabilities, indicating the uncertainty that some anti-pattern indeed occurred. After that, a focused manual inspection is needed.

Perkusich et al. [92] developed a procedure for modeling the whole process of a project as a BN to detect problems (bottlenecks) inside the project that can afterwards become a focus of corrective and preventive SPI methods and thus increase the chance of a successful project completion. The procedure first requires the construction of the BN for the specific process, meaning the identification of key process factors and the quantification of their relationships. The researchers demonstrated their approach on a Scrum-based development projects, validating the constructed BN through simulated scenarios and the procedure in two development projects. In this study they used human input only to construct the BN and are currently working on a mechanism based on metrics.

The approach developed by Smiari, Bibi and Stamelos [109] provides a framework for acquiring knowledge during software project development and models it through anti-patterns represented by BNs. They start with a generic BN that models application development, which can be tailored to the needs of the individual development environment. Through Bayesian analysis they measure people, process, project and product to support PM decisions providing and illustrate the effects of uncertainty by taking advantage of the nature of BNs, which provide a graphical representation of the probabilistic relationships among the set of variables.

### 2.7.5 Others

Palma et al. [90] detect business process anti-patterns via services, first using domain specific language to specify the anti-patterns based on rules, then generating the detection algorithms and finally detecting the anti-patterns in structured business processes (e.g., in BPMN). In the future they strive for automated generation of detection algorithms to not have to implement the detection manually for each new anti-pattern.

The concept of meta-patterns [123] works with the idea of building high-level patterns from smaller, "basic" patterns as components. The approach is used by researchers in the domain of object oriented programming [113] and data mining [121]. While not applicable for the current stage of our research, we will consider this approach in future stages of our work.

David Johnston [62] and Ian Mitchell [80] provide comprehensive work on agile anti-patterns and patterns, respectively, their recognition, usage and handling.

## 2.8 Quality Measuring

A realm of software project productivity and product quality measuring is a well researched area with many validated and suggested data-based objective metrics included. Sources cover wide range of topics form software development metrics in general [84] to software process management measuring [36] to metrics for specific methodologies (e.g., agile [29]), or specific types of software (e.g., [93]).

However, selection of appropriate metrics and their adjustment to fit the context of a specific project can prove to be a challenge. Kitchenham et al. [66] proposed an adjusted size to effort ratio, instead of a simple size to effort ratio, to be used in projects with multiple possible size measures. The adjusted size aggregates multiple size measures with a regression parameter. The authors presented an example using the metric for web application projects and explained the relationship between effort prediction models and productivity models.

Of course, data accuracy is also an issue when measuring project and product quality. The authors of [52], for example, found 33,8% of defects (or bugs) in more than 7000 tickets from five OSS projects to be misclassified. Meaning, that the issues reported as bugs in 33,8% where in fact new features, documentation updates, refactoring issues, etc. This, obviously, skews any metric using the defect count, and shows that the data from ALM tools does not have to accurately reflect reality. The specific problem of issue classification is a topic undertaken e.g., by PaStA tool (see Section 2.4.4).

In the previous section, the use of GQM approach was mentioned when representing and detecting (anti-)patterns. As project and product quality metrics in essence represent patterns in project data, this approach is also usable for their measuring. This approach was used by Morasca and Russo [82] for a study of productivity in a real-life environment in the Italian Public Administration.

Finally, the threshold values for the metrics can also vary based on project context. This includes methodology used, scope and domain of the product.

This is exemplified in [21], where authors present an approach to identifying threshold values specifically for safety critical systems.

# Chapter 3

# Concept of the Thesis

This chapter presents the concept of our research, its main goals, considered and selected methods of achieving them, and describes current state of our efforts.

## 3.1 Target audience

Because of the intended main usage and benefits of our work described in the following section, we see the people in position of certain level of authority in the software development teams (such as team leads and project managers) as the main beneficiaries of our research. Although, due to the potential in the study of anti-patterns and their effects on projects the whole of SE community and researchers especially might find our research interesting as well.

## 3.2 Overview and Basic Concept

The approach we propose and the experimental tool for its validation (currently in development) is called **Software Process Anti-patterns Detector (SPADe)**. Its central idea is to mine data from software project repositories in various ALM tools and analyze it in regards to the potential presence of patterns, reoccurring events, behaviors, concepts and methods, with special focus on anti-patterns, patterns whose occurrences have a negative effect on the project and its product. The main motivation is to combine the potential of vast volumes of project data stored in ALM tools and the theoretical guidance on processes, practices and (anti-)patterns in an automated way to improve the chances of projects to succeed and enhance the knowledge in the field.

The basic architectural concept of the approach is captured in Figure 3.1 and will be briefly described in the following paragraphs. Similarly focused existing research is described in Section 2.4 along with points of divergence from ours, and reasons why each given approach is incompatible with and unusable for reaching our goals.

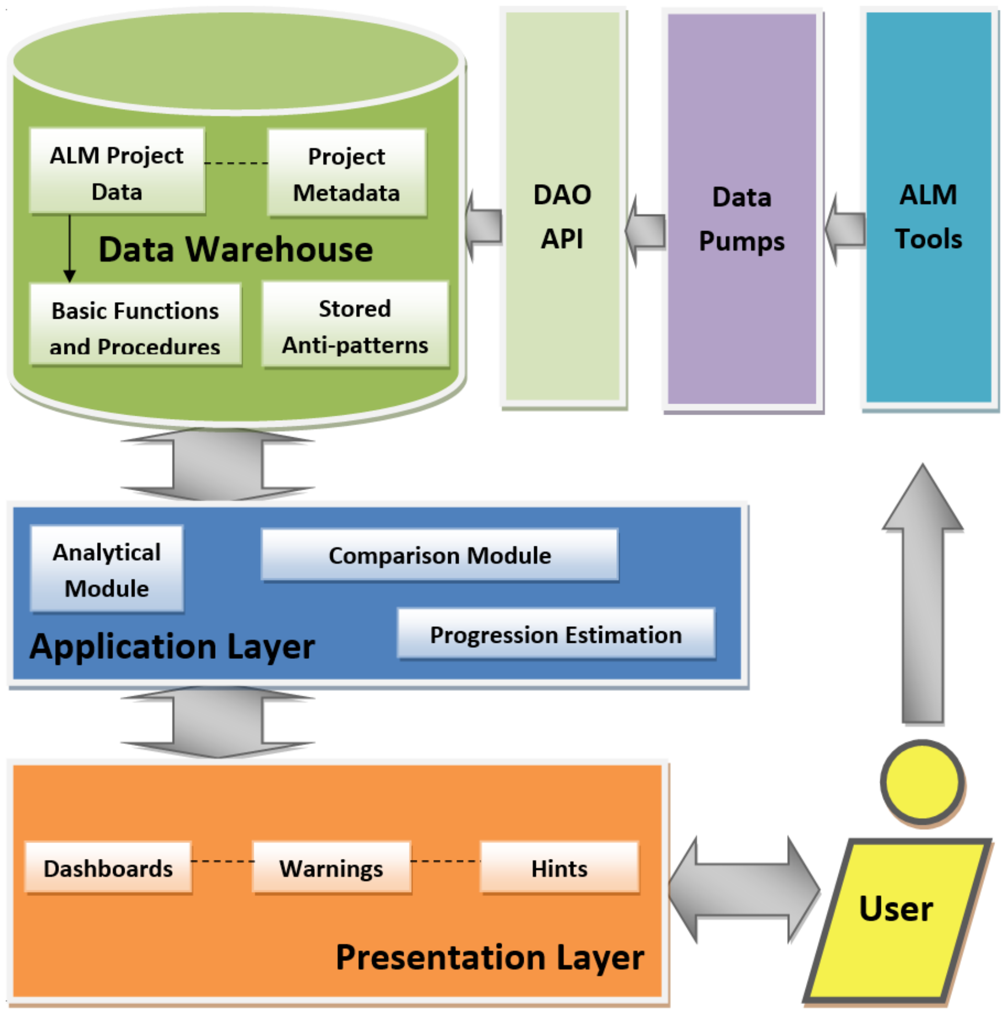


Figure 3.1: Overall architectural concept of SPADe

Software development projects store their data in the ALM tools for better management, resource usage efficiency, monitoring, communication and coordination. Meaning the tools hold significant data about current status and history of each such project. To harness the potential in this data, we first need to recover it. This will be done through data pumps, simple ETL software tools, one for each mined tool.

To be able to analyze this data, not only immediately after the mining process, but also retrospectively, and also to potentially compare data from different projects, we need a storage space. A data warehouse or database will therefore be used. But, because we intend to mine not only data from different projects, but also from different tools with the intention of unified way of analyzing it, the data warehouse has to store data in a unified format, independent of the source tool. Depending on the technology used, the data warehouse itself can calculate and store basic statistics like sums, averages, maximums and minimums, etc. with stored functions and procedures. The warehouse will also store the detectable patterns and anti-patterns and metadata about the project, like size, timespan, methodology or process it utilizes, type (brown-/green-field), product type (e.g., web/standalone application), etc. This is because some patterns might be specific to projects with certain attributes. For example, there is no point in looking for a GUI prototype in project developing command line interface (CLI) application, or for signs of sprint retrospectives in project with Waterfall process.

After successfully obtaining and storing the data, the application layer can then perform different kinds of analyses. The main one being the (anti-)pattern detection, but others, like comparisons of similar project, estimation of progress based on the current state and emerging patterns discovery, are also viable options.

The results of the analyses can be presented in a dashboard-like GUI to the user, complete with warnings of detected anti-patterns, potential bad state of the project and hints for remedies.

The following sections describe potential approaches and our selected or devised ones for accomplishing this overall idea.

### 3.2.1 Main usage and benefits

The main focus of our research is detection of PM and process (anti-)patterns in ALM data of software development projects and their relation to product quality and project success/failure. The benefits include discovery of anti-patterns early, providing actionable evidence for course-correcting decisions, monitoring the adhesion to the prescribed or recommended process model or methodology, and evidence-based support for post-mortem analysis and SPI. Other potential usages are outlined in Chapter 4.

## 3.3 Universal Metamodel

To meet our goal of storing project data from ALM tools in a unified format (RQ1, see Chapter 1), which separates our work from e.g. CHAOSS research (see Section 2.4.1), we have investigated various means of software process modeling (see Section 2.5). None of them fit strictly the needs of our research for different reasons.

SPEM (see Section 2.5.1) covers much the same ground as our intended work. However, its main purpose is to describe a process divorced from actual realities of projects. For example, it only models which roles the people performing certain tasks should have and has no entities to capture the real people themselves. This is a problem because of the N:M relation between people and roles mentioned in Section 2.1.1. Tasks themselves lack several attributes usually captured in ALM tools such as actual spent time, start date, due date, resolutions, etc. Modeling iterations and phases is also problematic. Specifically the very real and possible situation when the reaching of milestone of a phase does not coincide with the end of an iteration, but rather occurs somewhere during its course. A way to capture criteria for reaching a milestone other than textual description is also missing. Furthermore, the concept of splitting an element (task, role or artifact) into its description and later usage instances which may or may not override the content of the description is of no use for our purposes and could make the mapping of the ALM data into SPEM model unnecessarily problematic, if not impossible.

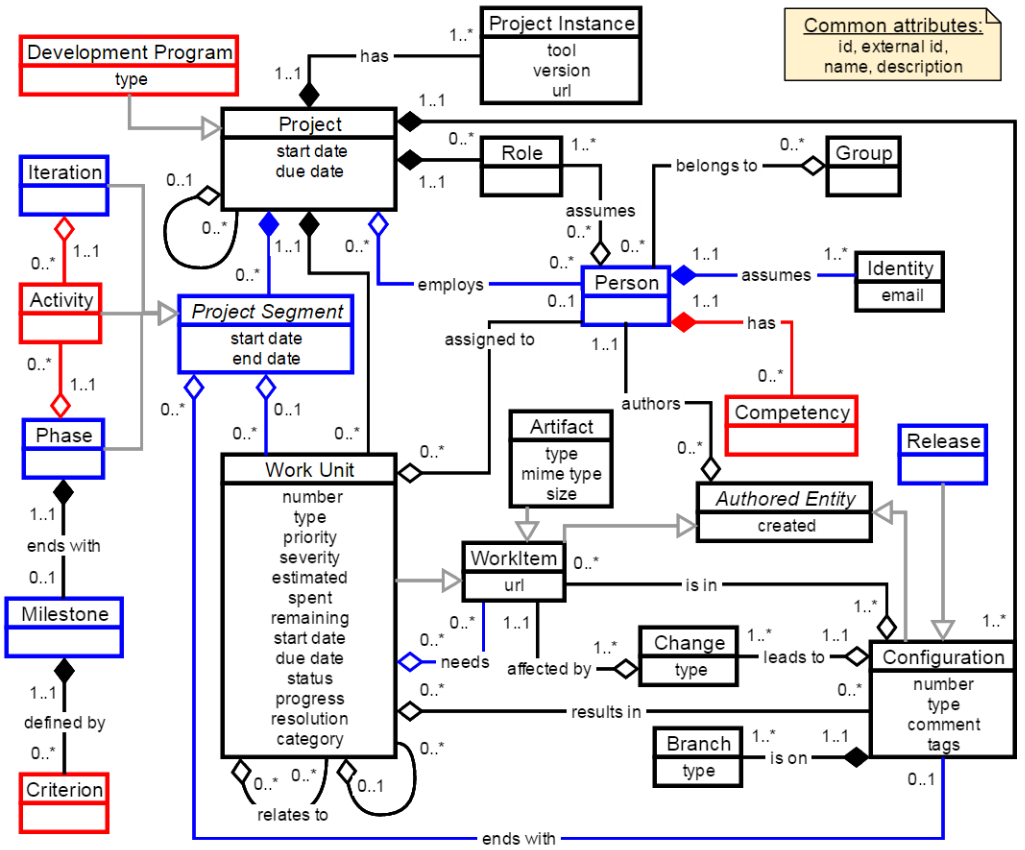


Figure 3.2: SPADe unified domain metamodel

OSLC (see Section 2.5.2) specifications, while focused specifically on ALM tools data, are currently not developed enough and cover only a small portion of data we need to collect in order to perform accurate analyses. Specifically, the Change Management specification is in stable versions 2.0 and 3.0 as a draft none of which covers all data we can collect and may need to analyze. Configuration Management exist as a draft only giving us even less to work with.

BPMN (see Section 2.5.5) was designed for modeling business processes. Therefore it uses many concepts not present in the ALM data (and therefore unusable for our purposes), while simultaneously lacking elements of ALM specific data we need to capture. The same problem of focusing on other specific domains than the ALM data disqualifies usage of ISO standards 24744 and 29110 also described in Section 2.5

Because of the above mentioned deficiencies (further described also in [95]) of the existing metamodels, we proposed our own [95] for storing the collected data. The domain form of the metamodel is shown in Figure 3.2 and the following sections describe it in greater detail. The color coding is explained in Section 3.3.4.

## 3.3.1 Construction

Several options have been identified for constructing the meta-model to be able to capture data from the selected toolset (see Section 3.4.1) while minimizing the possibility to lose data potentially useful for our intended analyses: (a) union, (b) intersection, (c) mixed.

The union approach would mean incorporating every single attribute captured by every single tool. The obvious benefit is the ability to store maximum of the available data. However, there are significant disadvantages. Not only is the approach overwhelming and needless (because not every attribute from every tool is necessarily useful for pattern detection) but also quite impossible. Because of the already mention customization capabilities of some tools (mainly those with issue-tracking component) all the possible custom attributes that every individual organization, team or person may incorporate into the data cannot be predicted. One possibility to circumvent this would be to capture the custom attributes as just the name and value pair. And while this is definitely an option we may revisit in the future to enhance the capabilities and accuracy of SPADe, for now it would be needlessly arduous a complicated to implement the analysis on such basis.

The intersection approach would mean storing only those entities and attributes present only in all tools[1]. The main disadvantage of this approach

[1]Obviously we do not mean the intersection of data from different types of tools.

is that it is limiting to the point of uselessness of the collected data. The reason is (among others) that GitHub stores only summary, description, assignee, target release, labels a simple open/closed status in tickets. Standard attributes from other tools like priority, estimate, severity, type, resolution, due date and others would have to be stored in e.g., description attribute in SPADe in a way conducive to parsing it during each analysis time and time again. Other example could be that SVN has only one person associated with a commit while in Git there is author, committer and other people signing off in different capacities on the changes presented by the commit.

Since both extreme cases proved unfeasible, we opted for the middle road – the mixed approach. We started with the intersection of all data captured in different tools and afterwards weight all the elements to decide if they should be included. The main criteria being the number of tools the element was present in and the potential usefulness to our analyses. We then enhanced the model with the passing elements as well as other concepts from e.g., software development methodologies we found necessary or useful for our purposes. The methodologies had to be considered because the metamodel needs to be able to capture development process irrespective of its model or methodology. That is the reason for including for example *Iteration*, *Phase* and *Milestone*, for which only some ALM tools have the corresponding concept.

Despite none of the existing models examined in Section 2.5 fitting our purposes exactly, there still is some measure of influence and inspiration we took from them and similar research efforts in Section 2.4. For example, we did make sure all the concepts from the current versions of Change and Configuration Management specifications of OSLC that would enhance analytical capabilities of SPADe where included in the metamodel.

After identifying the elements we wanted to capture in the SPADe metamodel, we also had to settle on the naming conventions. The names for similar concept vary somewhat among all the studied models and tools. We predominantly used to most common terminology from the studied work or from SE practice, with some exceptions. The term *Work Item* comes from IBM RTC, while *Work Unit* from CoSEEEK (see Section 2.4.5).Though the meaning is not always the same as in the source we took the term from, we adopted these terms because we felt they describe the particular concepts best. The actual meaning of the terminology is described in the next section.

Rather a composite of all data present in all VCS, all data present in all issue-trackers, etc.

### 3.3.2 Entities

The following describes all the entities in the SPADe metamodel shown in Figure 3.2, some of their attributes and some of the relations between entities as well.

#### Activity

- a group of interrelated tasks (*Work Units*) with greater common goal;
- can represent e.g., disciplines or practices (see Section 2.2);

#### Artifact

- configuration item, subject to configuration management and version control
- the term has been adopted from RUP (in SPEM it is used for one kind of Work Product)
- **type** (enumeration) – in the realm of ALM data it can be a file or a folder in the VCS repository, an uploaded file or a wiki page in the tool (in the future also e.g., emails);

#### Authored Entity

- superclass extended by entities for which author and timestamp of creation is stored (*Work Item* and *Configuration*);
- **created** (timestamp) – time and date of creation;

#### Branch

- development branch from VCS;
- **type** (enumeration) – trunk / master branch or other;

#### Change

- represents a change of a single *Work Item* at a point in time;
- **type** (enumeration) – addition, deletion or modification of an artifact;

### Configuration

- any state resulting from any *Work Item Change* or set of *Work Item Changes* (e.g., repository commit, file upload, ticket (*Work Unit*) or wiki page edit);
- an action of changing a *Work Item* or a set of *Work Items* (e.g., commit of multiple files) performed in a singular point in time by an individual;
- **description** (string) – in *Configuration*, a description attribute holds the comment associated with the action;
- **number** (integer) – revision identifier in VCS user interface (not necessarily the same as external ID);
- **type** (enumeration) – revision in VCS or other state (e.g., after editing wiki page or ticket);
- **tags** (string list) – VCS tags;

### Competency

- represents expertise, certification, education or any such asset a person has;

### Criterion

- a partial goal of a phase, an indicator of achieving its milestone;

### Development Program

- a group of related projects (e.g., series of products, application versions for different platforms, etc.);

### Group

- can represent a team, an organization, or another subgroup of the people involved in the project;

#### Identity

- represents user account in a particular tool, since a project can use more than one tool for ALM purposes;
- **name** (string) – represents login or username;
- **description** (string) – represents full name;

#### Iteration

- repeatable sub-process from iterative and agile methodologies (see Section 2.2);

#### Milestone

- a get of goals to be met by the end of a development *Phase* (see Section 2.2);

#### Person

- the consolidation of identities on this entity allows for statistics of a particular person throughout various projects and tools used to be calculated;

#### Phase

- a segment of the project lifecycle from software development methodologies (see Section 2.2);

#### Project

- a set of *Project Instance* instances representing the whole project record from various source tools;

#### Project Instance

- holds information about a particular ALM tool installation, from which the project data were obtained;
- **url** (string) – source link of the project data;

**Project Segment**

- a superclass for sub-processes functioning as *Work Unit* containers (*Iteration*, *Phase*, *Activity*);

**Release**

- release tagged in VCS or other important configuration;

**Role**

- a set of permissions and responsibilities a person can take on in the project;

**Work Item**

- superclass for both *Artifact* and *Work Unit*;
- **url** (string) – direct link to the actual *Work Item* (in repository, wiki, ALM tool GUI, etc.);

**Work Unit**

- task, ticket or issue from issue-tracker;
- the term has been adopted from CoSEEEK (see Section 2.4.5) because it most accurately describes the use of the concept in our context and is detached from the ALM specific terminology;
- **number** (integer) – ticket identifier in issue-tracking tool (unique inside a project);
- **type** (enumeration) – bug, task, feature, enhancement, etc.;
- **estimated, spent** (real) – amount of time;
- **remaining** (real) – estimated remaining time;
- **status** (enumeration) – ticket lifecycle status (new, assigned, resolved, closed, invalid, etc.);
- **progress** (integer) – percentage of the ticket effort done.

The self-referencing associations on *Project* and *Work Unit* represent that they have sub-projects or sub-units respectively. Other non-labeled associations represent the “contains” relationship. There is no relation between *Phase* and *Iteration* because a milestone achievement (end of a phase) does not have to correspond with the end of an iteration.

### 3.3.3 Enumerations

Apart from the entities, attributes and terminology, some enumerative values also vary throughout the toolset. This is an issue in attributes such as priority, severity, status, resolution and type in *Work Unit*, *Role* names and relation type between *Work Units*. Without a concrete set of values a the analysis cannot be undertaken in a unified way and comparison between projects with different enumeration value settings can hardly be done at all.

For example, lets say that as a part of anti-pattern detection we need to count the number of *Work Units* in “accepted” state. This status usually signifies that a submitted task has been reviewed and deemed valid and will be addressed by the team at some point in time. However, each tool (and due to customization in fact each tool installation) can use different term for this status. Furthermore, some tools may not have a value with this meaning available at all. A unification of such enumeration values is therefore needed.

Rather than simply unify the values, we opted for keeping the original value of each such attribute and at a classification schema on top of them. This way we can perform analysis in a unified way while not loosing the more granular and accurate original data. The classification schema was derived from the default values used by the selected tools while considering terms and distinctions from SE practice and methodologies. In some cases we used a two-tier classification schema, with class and superclass, for to enhance the capabilities and information provided by SPADe.

Following subsections describe the classification schema of each particular enumeration for which such schema was used. Each enumeration description is accompanied by a table showing the classes, the source tool(s) of the particular terminology, commonly used synonyms and mapping to superclasses. If not apparent the explanation of the classes and superclasses meaning is also given in the text.

Though it may appear some of the classes and/or terminology is not always taken directly from any particular source tool (at least not in their default setting), they do appear in many projects or have been taken from other sources, for example, software development methodologies (see Section 2.2). Examples are *Work Unit* relations and *Roles*. The reason is most often a consistency in terminology or the benefit we see in including these classes as a result of our mixed approach to constructing the overall metamodel (see Section 3.3.1).

**Priority and Severity** are enumerations with a lot in common. Both signify a degree of their respective measures, both typically consist of a medium value and several higher and lower values than the medium. The number of values up and down from the medium varies, but two values (either up or down) is usually the average. Therefore, on the higher level of classification (superclass) we used the medium, one degree higher and one degree lower, and the medium, two degrees higher and two lower on the lower level of classification (class). This allows as the option of performing analysis on several levels of granularity on the scale. We added the “Unassigned” value to both levels of classification for *Work Units* where the value has not been set yet. We used the most common terminology found in the ALM tools for naming convention. Tables 3.1 and 3.2 show the classification schema for priority and severity, respectively.

Table 3.1: Classification schema for Work Unit priority

| **Priority class** | **Used in** | **Synonyms** | **Superclass** |
|---|---|---|---|
| Unassigned | Redmine | | Unassigned |
| Lowest | Assembla, Bugzilla, Jira | | Low |
| Low | Assembla, Bugzilla, Jira, Redmine | | Low |
| Normal | Assembla, Bugzilla, Redmine | Medium | Normal |
| High | Assembla, Bugzilla, Jira, Redmine | | High |
| Highest | Assembla, Bugzilla, Jira | Immediate, Urgent | High |

Table 3.2: Classification schema for Work Unit severity

| **Severity class** | **Used in** | **Synonyms** | **Superclass** |
|---|---|---|---|
| Unassigned | Redmine | Unclassified | Unassigned |
| Trivial | Bugzilla | | Minor |
| Minor | Bugzilla | Small | Minor |
| Normal | Bugzilla | Common, Moderate | Normal |
| Major | Bugzilla | Big | Major |
| Critical | Bugzilla, Redmine | Blocker | Major |

**Work Unit type** represents the character of the task. It may be a defect in the implementation (**Bug**), a change request for existing feature (**Enhancement**), a request for a new functionality (**Feature**) or any other issue that should be addressed, for example, administrative, related to documentation, a call for a meeting, etc. (**Task**). The values and names of our classes (in brackets above) are again derived from standard values used in ALM tools, and once more the **Unassigned** value is added.

We currently only have one-tier classification for *Work Unit* type but are considering revising the schema once the researchers from the PaStA project (see Section 2.4.4) make their survey into the perceived classes of software changes by developers public. If their results present more possible values valid for our analysis we might take them as classes and move our current classes to superclasses.

**Status** enumeration values are very diverse among the ALM tools and their different installations. This is because different organizations and teams want or need to capture different sets of events along the workflow of a task. We therefore considered the points in lifecycle of a task we would need to be able to recognize in our analyses. The result reflects the most commonly used status values in ALM tools, in development methodologies and in SE practice. For the higher level of classification, we used the simple distinction (as used by GitHub) of open and closed tasks, because whether the task is still active or yet finished is the most basic information we can work with. The classification schema is shown in Table 3.3.

Table 3.3: Classification schema for Work Unit status

| **Status class** | **Used in** | **Synonyms** | **Superclass** |
|---|---|---|---|
| Unassigned | | | Unassigned |
| New | Assembla, Redmine | Backlog, To Do, Unconfirmed | Open |
| Open | GitHub, Jira | | Open |
| Accepted | Assembla | Assigned | Open |
| In Progress | Jira, Redmine | Reopened | Open |
| Resolved | Bugzilla, Jira, Redmine | Under Review, Test | Open |
| Verified | Bugzilla | Approved, Feedback | Open |
| Done | Jira | Closed, Fixed | Closed |
| Closed | GitHub, Jira, Redmine | | Closed |
| Invalid | Assembla | Canceled, Rejected | Closed |
| Deleted | | | Closed |

The meaning of the class values is as follows:

- **Unassigned** – not a usual case, because most tools automatically put a submitted task to an entry status of the workflow (e.g., "New"), but the possibility of corrupted data or non-standard behavior, and therefore the possibility of an unassigned status value cannot be dismissed,
- **New** – a submitted task,
- **Open** – an ex-post added value for *Work Units* in the *Open* status superclass for which, however, none of its (sub-)classes apply (i.e., does not express the same meaning),
- **Accepted** – team reviewed the ticket in regards to duplicity, validity and comprehensiveness of its description and found it relevant to address at some point,
- **In Progress** – the work on the task has begun,
- **Resolved** – the team member assigned to the task deems it finished and submitted to verification by higher authority
- **Open** – the task has been verified and (possibly) awaits some final formal activities and processes to take place before being closed definitively,
- **Done** – work on the task has been successfully completed and the task requires no more effort or attention,
- **Closed** – situation similar to *Open* but for the superclass *Closed*,
- **Invalid**[2] – rather than being accepted, the team found the task irrelevant from the start or at any point during the lifecycle (e.g., customer no longer needs the feature),
- **Deleted** – a status added for tickets referenced somewhere in the project data but now longer existing in the ALM tool.

Other commonly used statuses include *Assigned*[3] and *Reopened*. We decided not to use these because the information they convey can easily be obtained through other data. Specifically, the assignee attribute in *Work Unit* is not empty for the former, and there is a change of status from *Closed* superclass to *Open* superclass in the history records of the ticket for the latter.

[2]This situation is more commonly captured by the resolution attribute, but some tools still use the value or lack means of capturing the resolution.

[3]meaning an accepted task assigned to a person responsible for its completion

Table 3.4: Classification schema for Work Unit resolution

| Resolution class | Used in | Synonyms | Superclass |
|---|---|---|---|
| Unassigned | | Unresolved | Unassigned |
| Duplicate | Bugzilla, Jira | | Finished |
| Invalid | Bugzilla | | Finished |
| Won't Fix | Bugzilla, Jira | Won't Do | Finished |
| Works as Designed | | | Finished |
| Fixed | Bugzilla, Jira | Done, Fixed Upstream | Finished |
| Finished | | | Finished |
| Incomplete | Bugzilla, Jira | Cannot Reproduce | Unfinished |
| Works for Me | Bugzilla | | Unfinished |
| Unfinished | | | Unfinished |

**Resolution** specifies the circumstances of the ticket being closed. The main distinction being, if the task was finished or does not require further attention for another reason, or if some clarification or additional action is needed. This is the basis for the superclasses used. The process of deriving the classes was similar to the ones in status. The resulting classification is shown in Table 3.4 and the class values are:

- **Unassigned** – here more relevant than in status because many tasks do not have resolution value assigned until the ed of their lifecycle,
- **Duplicate** – a similar task has already been submitted,
- **Invalid** – similar to the status of the same name,
- **Won't Fix** – the team recognizes that the ticket (in this instance most probably defect) represents an existing problem but decides not to fix it (e.g., for its low severity or priority),
- **Works as Designed** – typical "it's not a bug, it's a feature" situation, when the reporter of the issue mistakes a deliberate behavior of the software for a defect
- **Fixed** – a fallback class for tasks in the similarly named superclass but not fit for any other class
- **Finished** – fallback class for units form similarly named superclass not fitting into any other class,
- **Incomplete** – the information in the task is incomplete or incomprehensible, making addressing the issue impossible,

- **Work for Me** – a team member was not able to replicate the issue (defect) described, calling for independent review by another person or more detailed context information,
- **Unfinished** – similar to *Finished* for its respective superclass.

**Work Unit relations** provide the capability to convey connection of several kinds between two tasks. Table 3.5 show the classification schema for relations. The types of relations most commonly found in ALM tools, and our superclasses, are described below.

Table 3.5: Classification schema for Work Unit relations

| **Relation class** | **Used in** | **Synonyms** | **Superclass** |
|---|---|---|---|
| Unassigned | | Unspecified | Unassigned |
| Duplicates | Jira, Redmine | Duplicate | Similarity |
| Duplicated by | Redmine | Duplicate(s) | Similarity |
| Copied from | Redmine | Clones | Similarity |
| Copied by | | Cloned by, Copied to | Similarity |
| Blocks | Bugzilla, Jira, Remine | | Temporal |
| Blocked by | Redmine | Depends on | Temporal |
| Precedes | Redmine | Before, Predecessor | Temporal |
| Follows | Redmine | After, Successor | Temporal |
| Child of | | Child, Subtask | Hierarchical |
| Parent of | | Parent | Hierarchical |
| Causes | Jira | | Causal |
| Caused by | Jira | | Causal |
| Resolves | | | Causal |
| Resolved by | | | Causal |
| Relates to | Jira, Redmine | Related | General |
| Mentions | | | General |
| Mentioned by | | | General |

- **Similarity** – one task is either a duplicate or a straight copy of the other,
- **Temporal** – one task should follow after or is outright blocked by another,
- **Hierarchical** – one task is superior to the other or to multiple others, which represent decomposed components of the first one,
- **Causal** – one issue either causes or resolves another.

- **General** – a simple connection between two tickets either as a explicit type of relation or through one mentioning the other (e.g., by the identifier of the latter in the description of the former).

Almost all the relations are non-symmetrical, meaning the nature of the relationship is different when viewed from each of its endpoints. The exception is the general relation which is equal from both sides. This, and the slight distinction mentioned in superclasses descriptions above (i.e., being a duplicate vs. being a copy, following vs. blocking, causing vs. resolving) led us to the set of values for the lower level classification.

**Role** of each person involved in a project can be, in our view, either taken straight from the ALM tool data (provided its present) or inferred through analysis of the activities the person takes part in and tasks he/she performs. The classes for the enumeration are derived again from the tools and software development methodology theory (e.g. [68]). On the higher level we would like to distinguish at least the member of the development team, the higher level of managerial roles, other stakeholders (customers, mentors, etc.) and outside people, or non-members of the organization. These often appear in projects, where at least the change management system is accessible for the public to be able to request or suggest changes to the software or report defects. The schema is shown in Table 3.6.

Table 3.6: Classification schema for Role name

| **Role class** | **Used in** | **Synonyms** | **Superclass** |
|---|---|---|---|
| Unassigned | | Everyone | Unassigned |
| Non-member | Assembla, GitHub | Anonymous | Non-member |
| Mentor | | Scrum Master | Stakeholder |
| Stakeholder | | Product Owner, Reporter, User, Watcher | Stakeholder |
| Project Manager | | Administrator, Manager, Owner, Project Lead | Management |
| Team Member | | Contributor, Member | Team Member |
| Analyst | | | Team Member |
| Designer | | Architect | Team Member |
| Developer | Jira, Redmine | Collaborator | Team Member |
| Tester | | | Team Member |
| Documenter | | | Team Member |

### 3.3.4 Absent Data Inference

Due to our process of construction of the SPADe metamodel described in Section 3.3.1 and the fact that specific data are necessary to perform some of our intended analyses, some of the data can not be transferred from the ALM tools in a straightforward fashion and require a degree of transformation. This can be caused by several factors, including absence of the data in the source tools, the need for restructuring some of it, or minimizing the reliance of our analysis on the disciplined use of the source tools by developers.

As an example, this issue also goes beyond the enumeration values classification described in Section 3.3.3. Unfortunately, not only do the tools use different terminology for the same task relations, but some use symmetrical records, whereas some do not. That means in some tools the relation is captured in both involved tickets and in some only in the source ticket of the relation. Some of these issues can to be resolved by analysis performed either immediately after or even during the data mining process. Some, though, go beyond the possibilities of pure inference or analysis performed on the data and require at least some measure of user involvement in order to obtain the most decisive, complete, consistent and reality reflecting information possible.

Different entities from the SPADe data model require different levels analysis and/or user input. The entities in Figure 3.2 in black require minimal involvement and can be for the most part mapped straight to the elements from ALM tools. The entities in blue require moderate additional effort, while the entities in red are in need of heavy analysis or user input. All these challenges can be in ideal cases resolved by parsing the textual data (wiki pages, ticket descriptions, commit messages, etc.) looking for mentions of other entities. Pattern matching based on regular expressions is one way of accomplishing this, provided the information is included in the data. Otherwise, the completion of data relies on user input. The specific challenges leading to the color coding in the metamodel figure are:

- detecting iterations and phases,
- distinguishing iterations from phases,
- detecting milestones,
- capturing milestone criteria to check their achievement,
- recognizing the releases in VCS data,
- linking iteration and/or phases releases with their appropriate project segments,

- detecting prerequisites of tasks,
- grouping projects into development programs,
- detecting and factoring in competencies,
- recognizing identities belonging to the same individual person.

At this stage of the research, the method(s) for addressing the above issues are part of the future work. For example, for the last one a simple heuristic approach can be implemented to compare usernames, email addresses and different forms of names (e.g., using full names or just initials for first names). Already implemented and validated approaches from related research can also be adopted (e.g., Section 2.4.4).

## 3.4 ALM Data Mining

The research into MSR was previously described in Section 2.6.1. This section covers the set of ALM tools we have selected for mining, the methods for MSR available and the limitations of mining these repositories.

### 3.4.1 Selected Set

Several criteria have been considered when selecting which ALM tools we want to mine data from. Among them, the general types (described in Section 2.3), the accessibility, the usage we observe in practice and the diversity in terms of capabilities and detail of captured information. We have decided to primarily focus on VCS, issue-tracking and full-fledged ALM tools, because they cover the other categories to certain extend as well. Many issue-trackers have wiki or other knowledge-base capabilities, the communication can be handled through commenting or in-built messaging systems and VCS and issue-trackers are often used even for capturing requirements and quality related artifacts and tasks. The inclusion of both VCS and issue-tracking tools data is what distinguishes our work from efforts aimed at only one type of tools, for example, Bloof (see Section 2.4.2). In our approach, where we include more than one tool from each type, we also differ from other research efforts, like SoftChange, Codeface, PaStA, YOSHI and Hipikat (see Section 2.4). The final set of tools we settled on is the following:

- **Git** and **SVN** – for their wide-spread use and the pair being the representatives of decentralized and centralized VCS tools respectively,
- **GitHub** – for its prevalence in ever-growing OSS development domain,

- **Bugzilla** – for its use by high-profile organizations like Mozilla and Apache,
- **Redmine** – which we use for educational and research purposes and is therefore a great source of smaller projects (useful for initial testing) and student projects (to contrast wit industry),
- **Jira** and **Assembla** – used by several of industry partners of our university, giving us a chance to analyze in-house projects and contrast them with OSS.

Other ALM tools, such as mail repositories will definitely be included in the future and once our ideas and their implementation is validated, the expansion of the toolset is virtually limitless. But we are confident the this initial set is sufficient to get to the intended results. Consolidated sources like Software Heritage and GitHub's experimental dataset (see Section 2.6.2) were also considered, but ultimately disregarded for now. The main reason validating the SPADe metamodel in regards to data from different source ALM tools.

### 3.4.2 Methods

There are basically four ways of mining ALM tools repositories:

1. direct database mining,
2. using APIs,
3. parsing exported data,
4. parsing web pages of GUI (a.k.a. web crawling).

Direct database access is rarely an option with the exception of servers we ourselves maintain. But, almost all ALM tool repositories provide some API, and certainly all from our selected set do. In all cases at least a REST API, in majority even API for Java[4] and other programming languages. The majority of tools also have some sort of data export capabilities in different formats, be it JSON, XML of plain text. The downfall of this approach is that often, due to tool-specific issues, the exports have to be done manually or requires extra steps and therefore more effort with hardly better results when compared to APIs. In early experimentation we found web crawling

[4]JGit, SVNKit, GitHub API for Java, Redmine Java API from TaskAdapter, Java wrapper around Assembla API from Matthew Sladen, Jira REST Java API from Atlassian, b4j for Bugzilla

somewhat unreliable, because the structure of the web pages differs between versions of the tool and even different server instances far more often than the APIs. In those instances, a modification of ETL data pumps can prove necessary for each individual server mined. Not to mention the not all tools have a web-based GUI (e.g. SVN and Git).

For all these reasons we focus on utilizing APIs as often as possible, only falling back on data exports and web crawling when proven absolutely necessary or for some unforeseen reason more straightforward than APIs.

### 3.4.3 Limitations

Many pitfalls of MSR generally [50] or for Git [14] and GitHub [63] specifically have already been outlined in literature. Here we will mention only a few important ones. From the standpoint of the approach itself, the most obvious issue are those of access, representativeness and incomplete data.

In the foreseeable future, possibility for any researcher to gain permission to access all the repositories of all the tools is hard to imagine, limiting any research aiming for large enough sample mostly to OSS projects. This can in turn lead to the practicing community rejecting the results as irrelevant for them. The divide between academia and practice is a long-standing issue, one of those that we identified in our review into problems in empirical SE research [94]. Even in instances, when researchers join with a private company and gain access to their data, it is usually to small a sample. Which leads directly to the second issue.

No matter how diverse or large a tool-set one selects, there is no guarantee that the projects mined will provide a representative sample of the industry in any measure. Moreover, the issue is made worse by the particular in-house projects that certain companies may allow the researchers to analyze and their relatively small number compared to the accessible OSS projects, returning back to the previous limitation.

Lastly, the data from many projects may prove incomplete, therefore skewing the results or rendering them useless completely. This may be cause by the lack of discipline or need of developers from the analyzed projects to document the progress and history of their efforts pushing the data captured further away from reality. Some organizations may not even use the ALM tools to their full or good enough degree, leaving out information vital for certain analyses, some of which may be impossible to infer otherwise. Even the capabilities of tools themselves may be limiting. For example, GitHub lacks the explicit capability to capture estimated and real efforts (i.e., time) spent on each particular task.

On the technical side, the issues hindering data mining and analyses include varying encoding, different standards of temporal data (timezones, timestamp format, etc.), and tools customization changing functionalities giving projects teams the option to capture same data in different (custom) attributes and making the mapping into unified format of project data more challenging.

## 3.5 Customization

As mentioned in Section 3.2 and shown in Figure 3.1, our approach needs metadata about the project in order to perform the analysis adequately. The metadata include the definition of the process or methodology the project should follow, its scale in terms of timespan and resources and the character of the product. Furthermore the metadata would include the mapping of the enumeration values used in the project to the classification schema described in Section 3.3.3 with the possibility to manually correct the automatically generated classification.

These factors can significantly skew the results of analysis if not taken into account. This is caused by the diverse nature of the software projects, the approaches of teams to using the ALM tools and difference in terminologies used, among other factors, and presents one of the bigger challenges of our research.

Similarly to the issues described in Section 3.3.4 some of the metadata can ideally be obtained from the ALM tools data itself (e.g., project timespan, staff size, process and product info from wiki), but if not, it needs to rely on the user input. This is especially the case for enumeration values and their mapping to our classification schemata. Unless the specific meaning of each value is present in the data (e.g., in some ALM tools policy document), or otherwise possible to infer, the mapping needs to be at least checked and, if needed, adjusted manually. The particular approach to collecting and inferring project metadata is subject of future work.

## 3.6 Anti-pattern format and detection

Though we intend on our approach to be applicable for general software process and PM pattern detection, our current main focus is to detect anti-patterns specifically. This does not take from the general usability of our approach because, as described in Section 2.1.5, we see anti-patterns as a subset of patterns, specifically consisting of those with harmful impact on project success or product quality.

This focus is unique to our research and different from similar work described in Section 2.4. Codeface focuses on community structures and evolution, PaStA on patches similarity, propagation and maintenance efforts, CoSEEEK on individual developer workflow and quality assurance tasks injection. SPARSE, PROMAISE and the work of Vranić et al. do deal with anti-patterns, but not their detection in actual project data. Rather, the former two focus on anti-pattern knowledge gathering and ontology and the latter on training developers in adopting practices and avoiding common mistakes.

To be able to detect anti-patterns in collected data (RQ2 and RQ3, see Chapter 1) we first need to compose the set of anti-patterns already defined and well-known in the practice. To this end, we performed a literature review and collected a set of anti-patterns presented in Table A.1 in Appendix A along with alternative names and sources in which they were found.

Some of the anti-patterns included stand on the border of the PM and e.g., architectural domain (JAR Hell, DLL Hell), etc. The reason is that the field of PM anti-patterns is relatively immature and the borderlines with other pattern domains are blurred. Other reason may be the fact, that PM itself interfaces heavily with other disciplines in software development and therefore total separation will never be possible. Nevertheless, even these bordering anti-patterns can be viewed from the PM and process perspective, hence being included by the authors of the sources.

The set of defined anti-patterns from literature is further expanded by several anti-patterns we garnered form experience in both education and industrial settings. These anti-patterns, each with a short description, are listed in Table B.1 in Appendix B. We will investigate the validity of these anti-patterns as part of future work.

Because the domain of PM anti-patterns is still changing, we started creating a catalogue with all gathered anti-patterns from literature and experience and make it public as an additional contribution of our work [16][5]. We hope this will lead to feedback, additions and corrections from researchers and practitioners, and foster creation and contributions to the PM pattern communities similar to the ones in design patterns domain [51].

## 3.6.1 Structured Description

Table 3.7 presents a more detailed definition of one of the anti-patterns we have called Collective Procrastination as an example. It also presents the structure we intend to use for textual description of the anti-patterns (e.g., in the aforementioned catalogue). It includes attributes commonly used in

[5]The catalogue is available at https://github.com/ReliSA/Software-process-antipatterns-catalogue

textual descriptions of anti-patterns (see Section 2.7.1): name and aliases of the anti-pattern, its short summary and symptoms. The "Sources" section lists the literature in which a description of the particular anti-pattern was found. The "Specific To" section lists the types of projects the particular anti-pattern is related to.

Table 3.7: Collective Procrastination – anti-pattern from experience

| **Name** | Collective Procrastination |
|---|---|
| **Also Known As** | |
| **Summary** | The team needs to handle several commitments in parallel, focuses on most pressing issues until that becomes the iteration, then rushes to finish it off. From the outside, actual status of work done is unknown during the iteration, hindering planning and problem solving. |
| **Symptoms** | rock-edge (one occurrence spanning the whole project) or staircase (repeated occurrence usually coinciding with a meeting, milestone or release date) burndown |
| | progress stalled for a time, then sudden increase |
| | a dramatic and fast paced (quasi-immediate) burst of issues solved prefaced by a stagnation period |
| **Specific To** | - |
| **Related Anti-patters** | Fire Drill – specific cause of management spending to much time on pre-development activities |
| **Sources** | experience |

As mentioned before, some anti-patterns are relevant only to projects with specific context. For example, only those using Scrum methodology [34, 33], RUP [70], or to innovation projects specifically [9]. Lastly, related anti-patterns are also listed. This can include more generic or specific instances of the same anti-pattern, opposite extremes of the same bad practice, patterns sharing several similar symptoms, etc. On top of added informational value, capturing relations among anti-patterns will foster reuse of components of their operationalization form, which will be described in the next section.

Further textual specification can be found in referenced sources for literature-sourced anti-patterns, though the ones from experience may need additional context and other sections to fully describe the anti-pattern and facilitate comprehension. The operationalization of each anti-pattern is described separately and is discussed in the following section.

At this stage in our work, only two of the anti-patterns from experience are described in at least the level of detail shown in Table 3.7. Namely Collective Procrastination and Nine Pregnant Women. The description of the latter is shown in Table C.1 in Appendix C. The pair was selected based on their fitness for detection in student projects which are easy to obtain, independence of project and product type, process model and other project metadata. All these aspects make them primary candidates for first experiments to provide preliminary validation of our approach.

## 3.6.2 Operationalization

The textual representation of the anti-patterns is not fit for automatic detection in data. We, therefore, need to express it in formalized representation (i.e., in our case using the SPADe metamodel; see Section 3.3) and consequently describe them in terms of data gathered from ALM tools. This process is called operationalization and is a precursory step towards detection itself [96].

One section of the textual representation decomposes anti-patterns into **symptoms** (see Section 2.7.1 and Table 3.7). These symptoms indicate the presence of the anti-pattern and are much easier to translate into terms of ALM tools data. So let there be $S$, a space of all possible symptoms. Then, each (anti-)pattern $p$ can be modelled as a subset of $S$:

$$S' \models p,\ S' \subseteq S.$$

To avoid confusion in terminology, we call symptoms expressed in terms of the SPADe metamodel entities **indicators**. Let $I$ be a space of all indicators. Then,:

$$\forall s \in S\ \exists\ I' \subseteq I,\ I' \models s,\ I' = \{i_0, i_1, i_2, \dots i_n\},$$

where $i_0, i_1, i_2, \dots i_n$ are individual indicators necessary to be detected (i.e., triggered; see Definition 2 in Section 3.6.3) in order to detect symptom $s$.

Once expressed in the SPADe metamodel terminology, the indicators are further decomposed into a set of measurements $G'$ of numerical and logical values, which is a subset of $G$, a space of all such measurements possible in SPADe data:

$$\forall i \in I\ \exists\ G' \subseteq G,\ G' = \{g_0, g_1, g_2, \dots g_m\}.$$

Numerical values (i.e., metrics) represent measures of specified entities or their attributes and arithmetic operations on them. Logical values signify presence/absence of some aspect of data (e.g., a textual attribute fitting a regular expression, existence of relation between given entities, etc.). Therefore,

$$\forall g \in G,\ g : \mathit{project\_data} \longrightarrow \mathbb{R}\ \cup\ \mathbb{B}.$$

This allows for one metric to be used in many indicators.

From all the above, we arrive at the definition of operationalized anti-pattern:

**Definition 1** (Operationalized pattern) Let $P$ be a space of all process and PM patterns, $S$ a space of all pattern symptoms, $I$ a space of all indicators expressed in project data terms and $G$ a space of all metrics (numerical and logical) measurable on project data from ALM tools. Operationalized pattern $p$ is then a set of metrics $G'$ on project data which map to the symptoms of the pattern:

$$p \in P,\ S' \models p,$$
$$\forall s \in S'\ \exists\ I' \models s,\ I' \subseteq I,$$
$$\forall i \in I'\ \exists\ G' = \{g_0, g_1, g_2, \dots g_m\},\ G' \subseteq G,$$

or simply put:

$$G' \models p.$$

### 3.6.3 Detection

The above model of (anti-)pattern operationalization is amenable for various technical reifications. In our case, data is stored in relational database management system (RDBMS) and the measurements on it have the form of SQL queries. The indicators, and therefore, (anti-)patterns themselves also have a form of (sets of) SQL queries in a script.

The detection depends on functions comparing the measured values with configured thresholds. Each metric $g$ is compared to its designated thresholds $t$ by its specific function $h$, such that:

$$h(g,t) : G \times (\mathbb{R} \cup \mathbb{B}) \to \mathbb{B}\ ,$$

and it reaching the threshold is signified by:

$$h(g,t) = \mathit{true}.$$

When all the metrics from a set representing a given indicator reach their respective thresholds, we say the indicator is **triggered**. A pattern is detected when all indicators necessary for its detection are triggered.

**Definition 2** (Triggered indicator) Let $tr : I \to \mathbb{B}$ be a binary function expressing the triggering of an indicator $i \in I$ represented by $m$ metrics

$$g_0,\ g_1,\ g_2,\ \dots\ g_m, \forall g_j \in G',\ G' \models i$$

compared by their specific functions $h_0, h_1, h_2, \dots h_m$ to their respective thresholds $t_0, t_1, t_2, \dots t_m$ :

$$tr(i) = \bigcap_{j=0}^{m} h_j(g_j, t_j),\ \forall g_j \in G'\ .$$

An indicator is triggered when $tr(i) = true$.

**Definition 3** (Detected pattern) Let $d : P \to \mathbb{B}$ be a binary function expressing detection of a pattern $p$ by its $n$ indicators

$$i_0,\ i_1,\ i_2,\ \dots\ i_n, \forall i_k \in I',\ I' \models p\ ,$$

such that:

$$d(p) = \bigcap_{k=0}^{n} tr(i_k) = true,\ \forall i_k \in I'\ .$$

A pattern $p$ is detected when $d(p) = true$.

### 3.6.4 Example – Collective Procrastination

As a validation of this pattern detection method, we have created an operationalized form of the Collective Procrastination anti-pattern (structured description shown in Table 3.7) in the form of an SQL script (see Appendix D) and detected the sought instances in the students projects data.

The symptoms shown in Table 3.7 can be summarized into two conditions: (1) a period of little to no activity (in terms of reported work done) occurs – we call this phenomenon "silence" – (2) and is followed by a period of abnormally dense effort, which we call a "cliff". An occurrence of the anti-pattern is demonstrated in the burndown chart shown in Figure 3.3. On the "total" line, depending on the calibration, the whole period up until April 9th is an instance of the "silence" phenomenon. The days from then until April 13th are an example of the "cliff" phenomenon.

The scope of the anti-pattern is an iteration, because monitoring the whole project can skew the results. For example, team can work on tasks not planned for current iteration, which is still procrastination. But, if viewed from the perspective of the whole project, it would disrupt the silence and cliff phenomena detection.

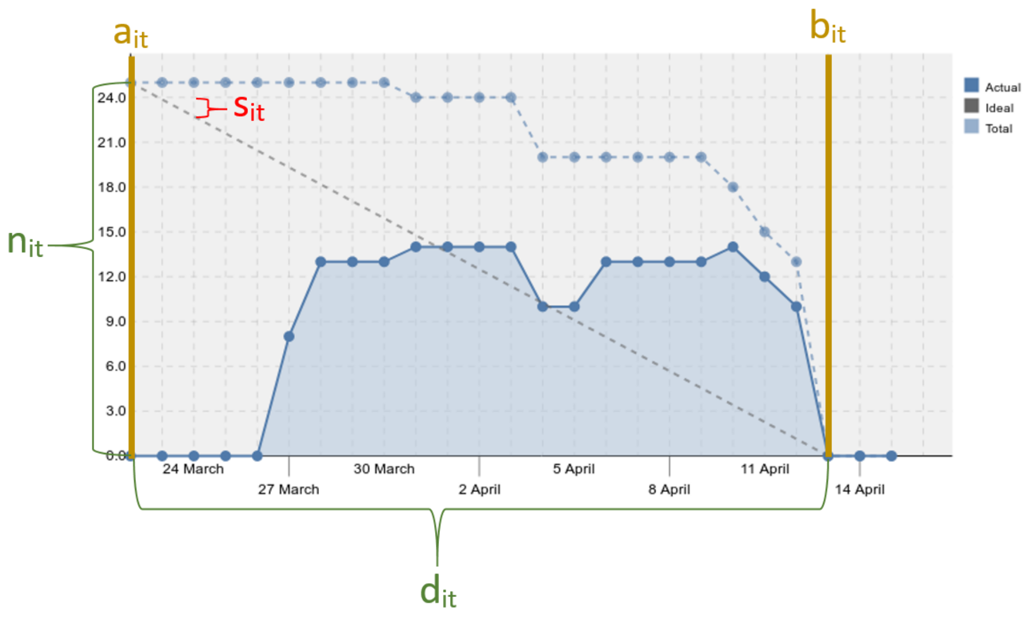


Figure 3.3: Burndown chart with calculated values

The detection process calculates the following metrics for each iteration *it* in the project:

- $a_{it}$ – start day of the iteration,
- $b_{it}$ – end day of the iteration,
- $d_{it} = b_{it} - a_{it} + 1$ – duration of the iteration in days,
- $n_{it}$ – number of tickets[6] in iteration,
- $s_{it} = n_{it}/d_{it}$ – average expected number of tickets closed per day[7].

The values are demonstrated in the burndown chart in Figure 3.3.

The symptoms can be expressed as indicators (i.e., in terms of SPADe meta-model):

1. $\mathbf{i_1}$ (potential "silence") – represented by metric $n_-$ which is the number of issues from an ALM tool within the iteration that changed their status to any value similar to "closed" during a period of $p_-$ days; the corresponding threshold is $t_1$;
2. $\mathbf{i_2}$ (potential "cliff") – represented by metric $n_+$ which is the number of issues within the iteration similarly changed (i.e., closed) during the

[6]also reffered to as issues, in SPADe terminology they are called *Work Units*
[7]the "actual" line in burndown chart

period of $p_+$ days directly following the $p_-$ period; the corresponding threshold is $t_2$.

The values $p_+$, $p_-$ are parameters which need to be set.

The metrics $n_-$ and $n_+$ each constitute a singular metric ($g_1$ and $g_2$, respectively) for the indicators $i_1$ and $i_2$, respectively. The detection criteria for the indicators are then:

1. $tr(i_1) \Leftrightarrow h_1(g_1, t_1),\ h_1 = (g_1 < t_1) = (n_- < t_1)$ and
2. $tr(i_2) \Leftrightarrow h_2(g_2, t_2),\ h_2 = (g_2 > t_2) = (n_+ > t_2)$ .

The Collective Procrastination anti-pattern ($p$) is detected if

$$d(p) \Leftrightarrow tr(i_1) \wedge tr(i_2) = true$$

for any day $x \in [a_{it}, b_{it}]$, where $it$ is the inspected iteration.

The actual metric values are obtained as follows. For each day in the iteration ($\forall i \in [a_{it}, b_{it}]$) we calculate:

- $n_x$ – the number of tickets from the iteration closed that day,
- $n_+$ – the number of tickets from the iteration closed in $p_+$ days, including the currently investigated, looking forward

$$n_+ = \sum_{j=x}^{min(x+p_+-1, b_{it})} n_j,$$

- $n_-$ – the number of tickets from the iteration closed in $p_-$ of previous days

$$n_- = \sum_{k=max(x-p_-, a_{it})}^{x-1} n_k.$$

The number of units closed during respective periods of time constituting silence or cliff (i.e., the thresholds $t_1$ and $t_2$) are apparently dependent on the size of the iteration, specifically, the number of issues planned for it. For instance, 10 closed tasks do not amount to much significance in an iteration with 250 tasks overall. But, they constitute a majority in an iteration of 15 tasks. Therefore, thresholds need to be adjusted for iteration size.

**Silence threshold** $t_1$ is the fraction of the overall number of tickets in the iteration low enough to qualify for the “little to no activity” in the first condition when applied to tickets closed in a day. It is $n_{it}$ divided by a “silence slow-down” parameter $p_s$, whose value needs to be set.

$$t_1 = n_{it}/p_s$$

Conversely, **cliff threshold** represents the steepness of the burndown line recognized as the "abnormally dense effort" in the second condition. To represent this, we must first multiply the expected daily average of closed issues $s_{it}$ by a "cliff speed-up" parameter $p_c$, whose value needs to be set. The resulting increased daily average must then be sustained over the period of $p_+$ days. Thus, the final number of issues closed serving as the "cliff" threshold is

$$t_2 = s_{it} * p_c * p_+$$

From all the above, we see that, to perform the detection process, we need to set the values for the above mentioned parameters $p_+$, $p_-$, $p_s$ and $p_c$. All of them are positive, non-zero integers:

$$\{p_+,\ p_-,\ p_s,\ p_c\} \subset \mathbb{N}$$

The rationale behind the values is the following:

- $p_-$ needs to be high enough to represent a significantly long period of silence and low enough to be able to detect the anti-pattern even in the case of week long iterations. Taking into account the possibility of occurrence of at least three-day weekends caused by national holidays, the value should be in the range

$$p_- \in [4, d_{it} - 1] \ .$$

- $p_+$ needs to eliminate instances of short (at least one- or two-days-long) bursts of effort in between two periods of silence and/or regular velocity of work. The upper limit has to leave a period in the iteration long enough to detect the previous silence period. Therefore,

$$p_+ \in [3, d_{it} - p_-] \ .$$

- $p_s$ needs to be high enough (and therefore, $1/p_s$ low enough) to be reasonably considered to represent slow pace in relation to the average tickets closed in a day ($s_i t$) and the length of the silence period ($p_-$). The silence threshold should be considerably lower than tickets closed with average velocity over the silence period, or

$$t_s \ll s_{it} * p_- \ .$$

  By using the above equations for $t_s$ and $s_i t$, we arrive at

$$p_s \gg d_{it}/p_- \ .$$

- $p_c$ needs to detect not the "return to regular tempo instances" but the panicky rush to make up for lost time.

$$p_c > 1$$

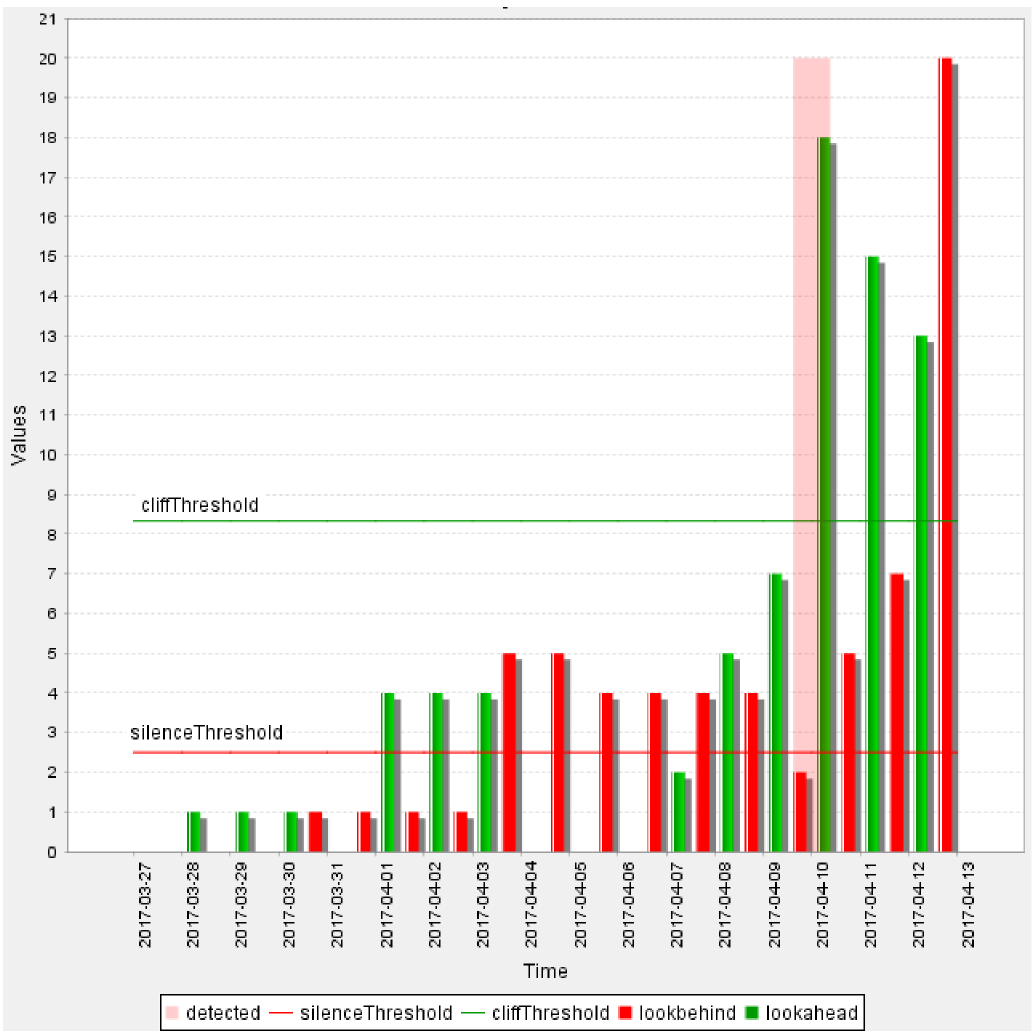


Figure 3.4: Collective Procrastination detected

For our preliminary experiments we have investigated 35 student projects which have iteration length of one to three weeks, and so we have selected the following parameter values:

$$p_+ = 3,$$

$$p_- = 6,$$

$$p_s = 10,$$

$$p_c = 2.$$

That means that in the last 6 days less than 10% of the overall tickets in iteration was closed and the average of tickets closed daily in the following 3 days needs to be more than double the ideal average.

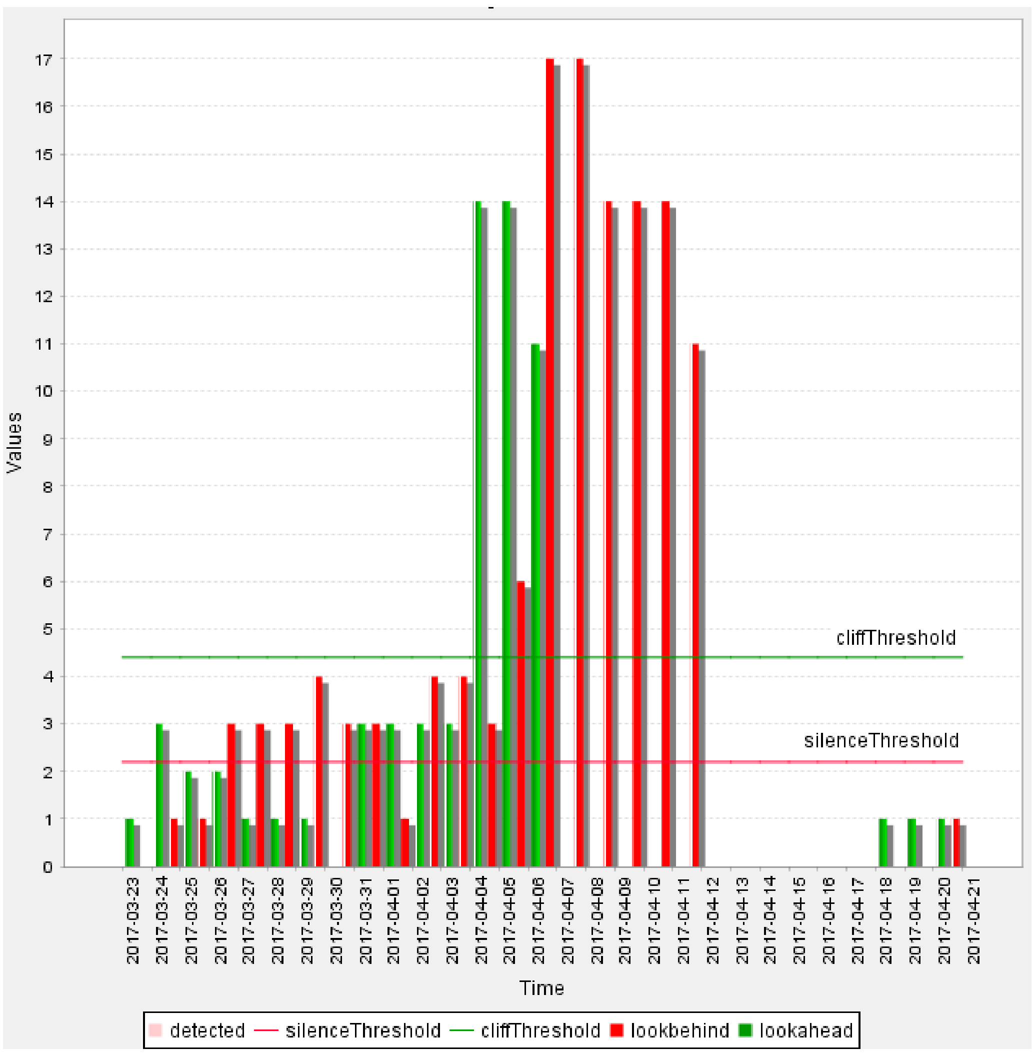


Figure 3.5: Collective Procrastination not detected – indicators

As is apparent from the rationale, the values not only may be, but definitely are dependent on the project context (metadata). Therefore, not only are these purely experimental values for preliminary validation of the overall approach, but will vary based on the analyzed projects.

Figure 3.4 shows the detection of the Collective Procrastination anti-pattern by our experimental tools in the iteration from Figure 3.3. In Figure 3.4 the **lookbehind** and **lookahead** values show the $n_-$ and $n_+$ values, respectively. From the chart it is apparent that detection criteria have been met only on April 10th, which is exactly where the silence ends and cliff begins in Figure 3.3.

Figures 3.5 and 3.6 show the burndown and detection charts, respectively, for another iteration where the anti-pattern was not detected. In Figure 3.6 the silence is present between March 27th and April 2nd but the subsequent cliff is not big enough. The cliff appears around April 5th and 6th, but is not preceded by silence with April 3rd breaking it. Consequently, there are dates in Figure 3.5 where lookbehind falls below silence threshold and lookahead rises over cliff threshold, but never both at the same time.

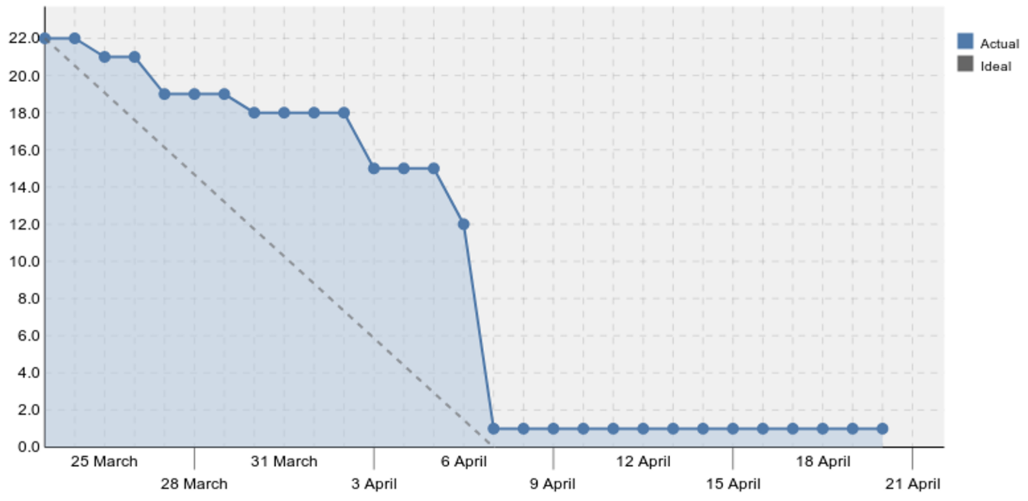


Figure 3.6: Collective Procrastination not detected – burndown

## 3.6.5 Advanced Approach

As mentioned in Section 3.6.1, some anti-patterns can be related. In extreme case, this can mean that one anti-pattern is part of another in its entirety, making it effectively the indicator. Conversely, some symptoms can have only one indicator and be considered as anti-patterns themselves in different context. The point being, there is no straight line between anti-pattern and indicator. The distinction can depend solely on the point of view. This allows for composition of indicators and anti-patterns into other anti-patterns, and is the reason for including information about related anti-patterns in our structure for their textual description (see Section 3.6.1).

Furthermore, some indicators can be used in other anti-patterns, only with different threshold values. The values can also vary based on the context of each individual project. The approach to operationalization should therefore allow for easy reconfiguration of the threshold values.

There may also be several sets of metrics (variants) to express the same indicator with the same validity. In fact, it is preferable to come up with such variants whenever possible. The reasons are mainly the incompleteness of data and inconsistencies in capabilities of different source tools (see Section 3.4.3). For example, a variant of an indicator can depend on overall time spent on an activity. But not all tools provide a time logging feature and, even if they do, not all project make use of it. Therefore, there is a benefit of having the option of detecting the indicator in another way.

Then, if indicator $i$ has $n$ variants $i_v$, it is triggered when any of its variants is triggered:

$$tr(i) = \bigcup_{v=0}^{n} tr(i_v).$$

The same applies to patterns because, as mentioned above, they themselves can be viewed as indicators in their entirety, and can therefore also have variants of the same validity. Pattern $p$ with $m$ variants $p_v$ is then detected when any of its variants is detected:

$$d(p) = \bigcup_{v=0}^{m} tr(p_v).$$

While the detection approach described in Section 3.6.3 is a good start for validating our research ideas, it leads to only binary (true/false) detection of anti-patterns (i.e., each metric reaches its threshold or not, the logical values are as expected or not, the anti-pattern is present or not). The complex reality of software development projects is rarely this black and white. Therefore, in the future a probabilistic approach based on e.g. BNs will be considered. The principle of decomposing anti-patterns into indicators and further into metrics is not too distant from the GQM method and the construction of BNs.

The function $h$ normalizing the distance of a metric $g$ to its threshold $t$ would then be:

$$h(g, t) : \mathbb{R} \times \mathbb{R} \longrightarrow [0, 1],$$

and the whole indicator $i$ composed of $n$ metrics could be reduced to a single value, like this:

$$i = \sum_{j=0}^{n} w_j * h_j(g_j, t_j),$$

where $w_j$ is the weight of the metric $g_j$ expressing its significance.

The rule for triggering indicator $i$ would then be:

$$tr(i) = (i > q),$$

where $q$ is a probability threshold for the indicator $i$.

Again, as mentioned above, patterns can be looked upon as more complex indicators. So, a pattern $p$ of $m$ indicators $i_k$, each with its respective weight $w_k$ and its probability threshold $q_k$ is detected if:

$$d(p) = \{[\sum_{k=0}^{m} w_k * (i_k - q_k)] > q_p\},$$

where $q_p$ is the probability threshold for the whole pattern.

The rule-based nature of comparing metrics to thresholds makes utilizing methods from similar research described in Sections 2.7.4 and 2.7.2 also a possibility. Further options of improving on the approach is the employment of Natural Language Processing (NLP) approaches for textual data and machine learning mechanisms.

The process of deriving the appropriate indicators from the textual description is presently reliant on manual execution. However, the need to streamline the process of operationalization of the anti-patterns for purposes of independently repeatable experimentation and eliminating the need to write the potentially extensive SQL queries manually for each individual (anti-)pattern compels us to consider options of partial automation. Research into automatic pattern extraction from processes exists (e.g. [42, 58, 100]), but is focused on mainly business processes using specific notations, like BPMN. However, the indicators do not have to be transcribed directly into the SQL query form.

We are also considering developing a specialized domain-specific language (for similar approaches see Section 2.7.2) for describing the indicators and anti-patterns which could be automatically processed. The language would be more readable for humans and the descriptions using it would be translated into SQL. Alternatively, we are considering using XML Schema Definition (XSD) schema to represent a pattern and XML data to represent the project, making use of the XSD validation of XML. The language for anti-patterns and indicators can then be a XSD schema that validates the anti-pattern XSD schemas.

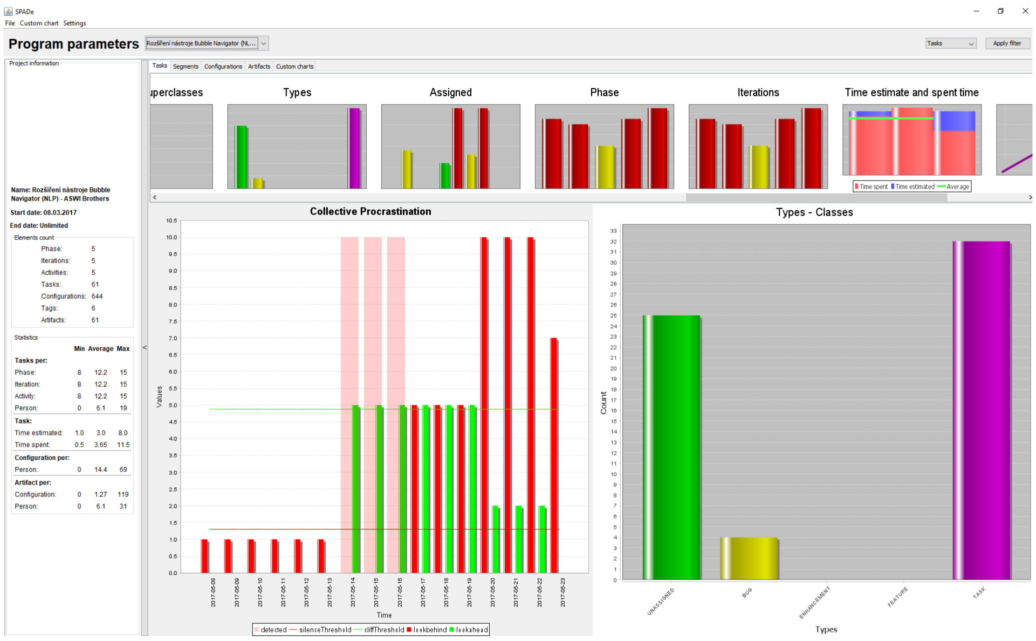
## 3.7 Presentation

The presentation layer of the experimental tool SPADe (see Section 3.2 and 3.1) has more functions than just enhancing the user experience with GUI base interface.

Data consistency during the transfer into SPADe database through data pumps is a major thread to validity of our approach. To mitigate this risk we need to be able to check the obtained data by comparing it the source tools data. This, however, is arduous and almost impossible effort when simply manually checking data in each table in the database after each instance of mining. Therefore, visual tools that facilitate easier validation of the successful transfer of the mined data in bulk are almost a necessity.

The visual presentation of the data can also serve to manually confirm the presence of a anti-pattern (or its symptoms) in the data (RQ3, see Chapter 1) before it is operationalized for automatic detection. This can in ideal circumstances lead even to identification and subsequent description of previously not considered anti-patterns.

We are currently developing several tools for such validation, examining different aspects of the data, such as aggregate metrics, charts, structures of and relations among entities, temporal distribution of developer actions and entities lifecycles. The prototype implementations of chart- and timeline-based GUIs are shown in Figures 3.7 and 3.8, respectively.


SPADe
File Custom chart Settings
Program parameters
Tasks
Apply filter
Tasks Segments Configurations Artifacts Custom charts
Project information
uperclasses
Types
Assigned
Phase
Iterations
Time estimate and spent time
Time spent Time estimated Average
Collective Procrastination
Values
Time
detected silenceThreshold cliffThreshold lookbehind lookahead
Types - Classes
Count
UNASSIGNED
BUG
ENHANCEMENT
FEATURE
TASK
Types

Name: Rozšíření nástroje Bubble Navigator (NLP) - ASWI Brothers

Start date: 08.03.2017

End date: Unlimited

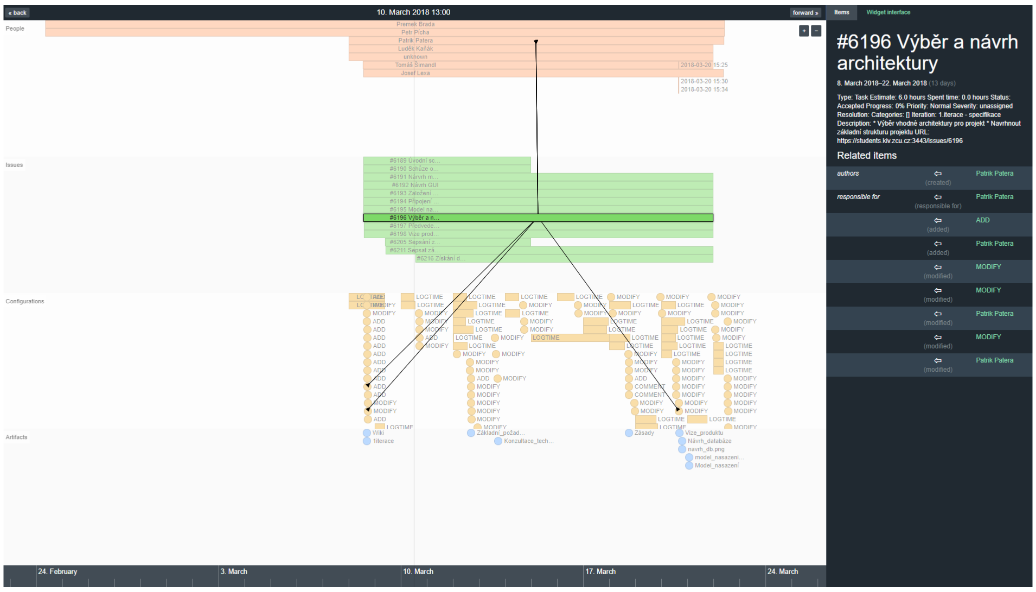
Elements count

| | |
|---|---|
| Phase: | 5 |
| Iterations: | 5 |
| Activities: | 5 |
| Tasks: | 61 |
| Configurations: | 644 |
| Tags: | 6 |
| Artifacts: | 61 |

Statistics

| | Min | Average | Max |
|---|---|---|---|
| **Tasks per:** | | | |
| Phase: | 8 | 12.2 | 15 |
| Iteration: | 8 | 12.2 | 15 |
| Activity: | 8 | 12.2 | 15 |
| Person: | 0 | 6.1 | 19 |
| **Task:** | | | |
| Time estimated: | 1.0 | 3.0 | 8.0 |
| Time spent: | 0.5 | 3.65 | 11.5 |
| **Configuration per:** | | | |
| Person: | 0 | 14.4 | 69 |
| **Artifact per:** | | | |
| Configuration: | 0 | 1.27 | 119 |
| Person: | 0 | 6.1 | 31 |

Figure 3.7: Chart-based GUI of SPADe



#6196 Výběr a návrh architektury

8. March 2018–22. March 2018 (13 days)

Type: Task Estimate: 6.0 hours Spent time: 0.0 hours Status: Accepted Progress: 0% Priority: Normal Severity: unassigned Resolution: Categories: [] Iteration: 1.iterace - specifikace Description: * Výběr vhodné architektury pro projekt * Navrhnout základní strukturu projektu URL: https://students.kiv.zcu.cz:3443/issues/6196

Related items

| | | |
|---|---|---|
| *authors* | ⇦ (created) | Patrik Patera |
| *responsible for* | ⇦ (responsible for) | Patrik Patera |
| | ⇦ (added) | ADD |
| | ⇦ (added) | Patrik Patera |
| | ⇦ (modified) | MODIFY |
| | ⇦ (modified) | MODIFY |
| | ⇦ (modified) | Patrik Patera |
| | ⇦ (modified) | MODIFY |
| | ⇦ (modified) | Patrik Patera |

Figure 3.8: Timeline visualization of SPADe data

Moreover, the SPADe data can be visualized in a node-edge type graph in Interactive Multimodal Graph Explorer (IMiGEr) [54], another tool being developed at our department (see Figure 3.9). IMiGEr is able to process and show data in an interactive interface provided they conform to simple, but specific format input file format. Specifically, it works with JSON files, which can represent either raw data, using a generic data format and limiting the functionality of the tool to an extend, or various domain specific data, which require a particular configuration first. The specific configuration for SPADe data has already been developed.

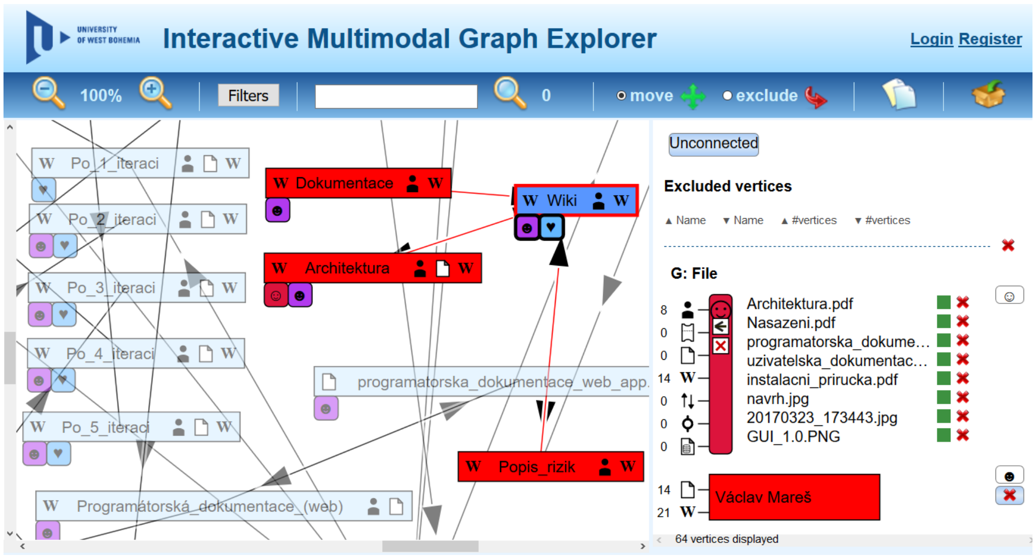


Figure 3.9: Visualization of SPADe data in IMiGEr [54]

## 3.8 Metrics

To be able to address the possible link between anti-pattern detection and product quality, as well as project success (RQ4, see Chapter 1), we have to measure these project aspects. A multitude of research has been done into software project and product measuring (see Section 2.8) and we will have all the relevant project data available through mining the ALM tools. This allows us to measure standard quality metrics, such as defect frequency in time or relative to the product size, adherence to a plan, testing efforts, average time to defect fix, remaining effort estimates, etc.

However, the challenge here is the selection of proper metrics both in general and considering the specific context of a project (an issue also mentioned in Section 2.8). Especially, if we want to compare different projects and their quality, we have to use similar metrics to not introduce unnecessary biases and threads to validity into our experiments. A related challenge is the determination of threshold values to be used.

The biggest issue in this aspect of our work is a certain measure of unreliability of the data from ALM tools. As shown in [52], the classification of tickets does not accurately reflect reality. And the same can be assumed in all other aspects of ALM data entities. The indication is, that the data has to be not only enhanced as described in Section 3.3.4 but cross-validated using as much of the available information inferred from the data as possible. The commit classification from PaStA project (see Section 2.4.4) should serve to resolve one of these issues, as it does not depend on the information from the developers (i.e., the assigned issue type) but rather on the resulting source code and commit message analysis. Similar mechanisms improving the data accuracy and minimizing the reliance on the developers input and the impact of misrepresentation of reality should be applied as much as possible.

Though not as objective as the project data itself, other approaches to gauging project and product quality, such as development team member and user surveys, can also be adopted. The results should nevertheless be confronted with the outcomes of project data analysis.

# Chapter 4

# Future Work

The main objective of our work in the near future is to implement enough of our approach into the SPADe experimental tool to be able to perform preliminary experiments through which we intend to validate our approach as a whole. While we will make an effort to obtain even data from in-house projects from our industrial partners, we will first focus on OSS and student projects data, which we have readily available. The main challenges after implementation therefore being the selecting a representative enough dataset and mining the projects, as well as transforming a substantive enough subset of anti-patterns into the operationalized form. The validation of the data transfer and the approach as a whole can also result in the need to expand the SPADe metamodel to include more detailed data as well as broadening the set of ALM tools mined, if it proves insufficient. Also, the metrics for product and project quality measurements will be established.

We already identified several research groups with whom we can work to mutually validate our approaches and enhance our respective research efforts. Some we reached an agreement with about our future collaboration, for instance, a research group from Chalmers University of Technology in Gothenburg, Sweden[78, 77] and Vranić et al. from Bratislava [38], some we already actively work with towards these goals [97]. An agreement on using our approach to validate the compliance of the process of our industrial partners with the process models they use (e.g., from standard EN 50128 [19]) in the future has also been reached.

## 4.1 Possible Additional Uses

We are convinced that our research can yield results and experimental tools with capabilities beyond our main intended usage described in Section 3.2.1. Some of the following ones were already mentioned throughout the text of this thesis concept.

The data pumps and unified metamodel for storing ALM data about projects can be used for e.g., data migration between different ALM tools[1]. Separate instance of the pumps and database can also be used by other researchers and practitioners to gather data from their own set of projects and perform either our anti-pattern detection, or any other analysis they need. Our dataset from OSS projects in the unified database can be made public to perform different analysis on by other users. Along with the anti-pattern catalogue, this may serve as a knowledge base for all SE communities.

The approach and implementation as a whole can be used for education purposes to showcase to students the mistakes they make, discussing the options of addressing them and checking their progress. Since a process can be described as a pattern and our anti-patterns are just harmful patterns, the approach can be used to check the adherence of a project to a particular process model, or to identify and describe the one used through practices (patterns) employed. The same can be done with different policies (e.g., for ALM tools usage, artifact content and format, etc.) used by company or team, which can be described as patterns, checked or identified (i.e., reverse engineered). We have already done the preliminary work on how to detect architects in the project data, or check the appropriateness of their activities in research conducted with our colleagues from OTH Regensburg [97].

Furthermore, the approach can be used for cross-project comparison, inferring the future progress estimation (i.e., probability of success, project being on schedule, product quality, etc.) of current project based on past similar projects and anti-patterns in common. Lastly, new (anti-)patterns can be discovered through studies of reoccurring phenomena in data from different projects, provided the data set is large enough.

[1]Provided both mining and pushing data capabilities are implemented in the pumps.

# Chapter 5

# Conclusion

This work mainly concerns the detection of commonly and frequently occurring bad practices (anti-patterns) in software development projects and their management in data gathered from ALM tools and the impact of these anti-patterns on project success and product quality.

The text has outlined the intended areas of our current work, described the background and existing research in similar fields throughout the SE community, presented the current state of our work, the adopted approaches and future research directions.

We have thoroughly researched current state of the art in areas of software development processes, ALM tools data mining, patterns and anti-patterns and measuring of software projects and products. Though research shows that work similar in some aspects to our own does exist, it also shows current state-of-the-art is unable to fully investigate the relation between PM anti-pattern occurrences in ALM project data and their impact on project success and product quality.

Finding no good enough match in existing software process metamodels, we have defined our own SPADe metamodel for storing project data from ALM tools independent of both the source tool and process model utilized by a particular project. We have implemented the database for ALM data based on our metamodel, the data pumps for various source tools and several GUIs for our experimental tool, SPADe. Through this implementation the feasibility of data mining from various ALM tools as well as the structure of the metamodel has been validated. We created two anti-pattern representations and performed experiments on their detection with some promising preliminary results. Along with the metamodel, the additional benefit of our work is the collection (catalogue) of PM anti-patterns enhanced with several patterns garnered from our own experiences in SE education and practice.

Based on the designed model and experiments performed we conclude that it is possible to (1) represent project data from different methodologies in unified format (answering RQ1), (2) represent and detect at least some subset of (anti-)patterns in the ALM data (partially answering RQ2 and RQ3).

Our work in the near future concerns steps to further validate our research, such as the fully operational implementation of SPADe, operationalization of selected anti-patterns, cross-validation with results from other researchers and examination of the relationship between anti-patterns detection and project and product quality metrics.

We are confident, that our work has significant potential in helping SE researchers and practitioners in understanding the realities of the software project development and the impact of PM missteps on software projects and products. Furthermore, we see great potential in utilizing different parts of our approach (dataset, metamodel, anti-pattern catalogue, SPADe implementation) for other research efforts, especially as it can be refocused from the narrower set of anti-patterns (i.e., harmful patterns) to patterns in general.

# References

[1] C. Alexander. *A pattern language: towns, buildings, construction.* Oxford University Press, Oxford, UK, 1977.

[2] S. W. Ambler. *Process patterns: building large-scale systems using object technology.* Cambridge University Press, Cambridge, UK, 1998.

[3] S. W. Ambler. The "Broken Iron Triangle" Software Development Anti-pattern. Online, 2012. http://www.ambysoft.com/essays/brokenTriangle.html. Accessed November 14th, 2018.

[4] S. W. Ambler. Common Role Anti-Patterns in Online Discussion Forums. Online, 2014. http://www.ambysoft.com/essays/discussionListAntiPatterns.html. Accessed November 14th, 2018.

[5] S. W. Ambler. The "Change Prevention Proces" Anti-Pattern. Online, 2014. http://www.ambysoft.com/essays/changePrevention.html. Accessed November 14th, 2018.

[6] S. W. Ambler and M. Lines. *Disciplined Agile Delivery.* IBM Press, Indianapolis, IN, USA, 2012.

[7] S. W. Ambler, J. Nalbone, and M. Vizdos. *The Enterprise Unified Process: extending the Rational Unified Process.* Prentice Hall Press, Upper Saddle River, NJ, USA, 2005.

[8] D. J. Anderson. *Kanban: successful evolutionary change for your technology business.* Blue Hole Press, 2010.

[9] D. Aydinli. Software project management anti-patterns in innovation projects. Master's thesis, University of Tampere, 2015.

[10] D. Aydinli, E. Berki, T. Poranen, and I. Stamelos. Management anti-patterns in IT innovation projects. In *Proceedings of the 20th*

*International Academic Mindtrek Conference*, pages 1–10, New York, NY, USA, 2016. ACM.

[11] L. Bass, I. Weber, and L. Zhu. *DevOps: A software architect's perspective.* Addison-Wesley Professional, Boston, MA, USA, 2015.

[12] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, et al. Manifesto for agile software development, 2001.

[13] K. Beck and E. Gamma. *Extreme programming explained: embrace change.* Addison-Wesley Professional, Boston, MA, USA, 2000.

[14] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining Git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10, Piscataway, NJ, USA, 2009. IEEE.

[15] S. Bourk and P. Kong. An Introduction to the Nexus Framework. Online, 2016. https://www.scrum.org/resources/introduction-nexus-framework. Accessed November 14th, 2018.

[16] P. Brada and P. Picha. Software Process Anti-pattern Catalogue. In *EuroPLoP '19: Proceedings of the 24th European Conference on Pattern Languages of Programs*, New York, NY, USA, 2019. ACM.

[17] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[18] W. J. Brown, H. W. McCormick, and S. W. Thomas. *Anti-patterns project management.* John Wiley & Sons, Inc., New York, NY, USA, 2000.

[19] BSI. Railway applications. Communication, signalling and processing systems. Software for railway control and protection systems., 2011.

[20] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A system of patterns: Pattern-oriented software architecture.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[21] Š. Cais and P. Pícha. Identifying software metrics thresholds for safety critical system. In *The Third International Conference on Informatics Engineering and Information Science (ICIEIS2014)*, pages 67–78, Kowloon, Hong Kong, 2014. The Society of Digital Information and Wireless Communications.

[22] G. Canfora, L. Cerulo, and M. Di Penta. Identifying Changed Source Code Lines from Version Repositories. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, volume 7, page 14, Washington, DC, USA, 2007. IEEE Computer Society Press.

[23] J. P. Castellanos Ardila, B. Gallina, and F. U. L. Muram. Enabling Compliance Checking against Safety Standards from SPEM 2.0 Process Models. In *The Euromicro Conference on Software Engineering and Advanced Applications*, Piscataway, NJ, USA, August 2018. IEEE.

[24] CHAOSS. GrimoireLab Tutorial. Online, 2018. https://chaoss.github.io/grimoirelab-tutorial/. Accessed November 14th, 2018.

[25] M. Cohn. Make the Product Backlog DEEP. Online, 2009. https://www.mountaingoatsoftware.com/blog/make-the-product-backlog-deep. Accessed November 14th, 2018.

[26] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the 25th international Conference on Software Engineering*, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society.

[27] W. Cunningham. Management Anti Pattern Road Map. Online, 2010. http://wiki.c2.com/?ManagementAntiPatternRoadMap. Accessed November 14th, 2018.

[28] W. Cunningham. Anti Patterns Catalog. Online, 2013. http://wiki.c2.com/?AntiPatternsCatalog. Accessed November 14th, 2018.

[29] C. W. H. Davis. *Agile Metrics in Action.* Manning Publications, Shelter Island, NY, USA, 2015.

[30] R. Di Cosmo and S. Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES 2017: 14th International Conference on Digital Preservation*, pages 1–10, 2017.

[31] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 131–136, Piscataway, NJ, USA, 2003. IEEE.

[32] R. Ellner, S. Al-Hilank, J. Drexler, M. Jung, D. Kips, and M. Philippsen. eSPEM – A SPEM extension for enactable behavior modeling. In *European Conference on Modelling Foundations and Applications*, pages 116–131, New York, NY, USA, 2010. Springer.

[33] V.-P. Eloranta, K. Koskimies, and T. Mikkonen. Exploring ScrumBut -— An empirical study of Scrum anti-patterns. *Information and Software Technology*, 74:194–203, 2016.

[34] V.-P. Eloranta, K. Koskimies, T. Mikkonen, and J. Vuorinen. Scrum Anti-Patterns – An Empirical Study. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, volume 1, pages 503–510, Piscataway, NJ, USA, 2013. IEEE.

[35] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 23–32, Piscataway, NJ, USA, 2003. IEEE.

[36] W. A. Florac, R. E. Park, and A. D. Carleton. Practical software measurement: Measuring for process management and improvement. Technical report, Carnegie-Mellon University Pittsburgh, PA, Software Engineering Institute, 1997.

[37] J. C. Freudenbegr. Certified Scrum Master Training. Technical report, Scrum Alliance, Inc., 2013.

[38] T. Frtala and V. Vranic. Animating organizational patterns. In *Cooperative and Human Aspects of Software Engineering (CHASE), 2015 IEEE/ACM 8th International Workshop on*, pages 8–14, Piscataway, NJ, USA, 2015. IEEE.

[39] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, Chennai, India, 1995.

[40] L. García-Borgoñon, M. A. Barcelona, J. A. García-García, M. Alba, and M. J. Escalona. Software process modeling languages: A systematic literature review. *Information and Software Technology*, 56(2):103–116, 2014.

[41] D. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd workshop on open source software engineering*, pages 63–67, Cork, Ireland, 2003. University College Cork.

[42] M. F. Gholami, P. Jamshidi, and F. Shams. A procedure for extracting software development process patterns. In *Computer*

*Modeling and Simulation (EMS), 2010 Fourth UKSim European Symposium on*, pages 75–83, Piscataway, NJ, USA, 2010. IEEE.

[43] Google. GitHub Data | BigQuery | Google Cloud. Online, 2018. https://cloud.google.com/bigquery/. Accessed November 14th, 2018.

[44] G. Grambow, R. Oberhauser, and M. Reichert. Contextual injection of quality measures into software engineering processes. *International Journal on Advances in Software*, 4(1&2):76–99, 2011.

[45] G. Grambow, R. Oberhauser, and M. Reichert. Event-driven exception handling for software engineering processes. In *International Conference on Business Process Management*, pages 414–426, New York, NY, USA, 2011. Springer.

[46] G. Grambow, R. Oberhauser, and M. Reichert. Towards a Workflow Language for Software Engineering. In *10th Int. Conf. on Software Engineering (SE'11)*, Calgary, Alberta, Canada, February 2011. ACTA Press.

[47] G. Grambow, R. Oberhauser, and M. Reichert. Automated software engineering process assessment: supporting diverse models using an ontology. *International Journal on Advances in Software*, 6(1 & 2):213–224, 2013.

[48] Z. Han, P. Gong, L. Zhang, J. Ling, and W. Huang. Definition and detection of control-flow anti-patterns in process models. In *Computer Software and Applications Conference Workshops (COMPSACW), 2013 IEEE 37th Annual*, pages 433–438, Piscataway, NJ, USA, 2013. IEEE.

[49] R. Hebig, G. Gabrysiak, and H. Giese. Towards patterns for mde-related processes to detect and handle changeability risks. In *Proceedings of the International Conference on Software and System Process*, pages 38–47, Piscataway, NJ, USA, 2012. IEEE Press.

[50] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. W. Godfrey. The MSR Cookbook: Mining a Decade of Research. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 343–352, Piscataway, NJ, USA, 2013. IEEE Press.

[51] S. Henninger and V. Corrêa. Software pattern communities: Current practices and challenges. In *Proceedings of the 14th Conference on Pattern Languages of Programs*, page 14, New York, NY, USA, 2007. ACM.

[52] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 2013 international conference on software engineering*, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[53] K. Herzig and A. Zeller. Mining bug data. In *Recommendation Systems in Software Engineering*, pages 131–171. Springer, New York, NY, USA, 2014.

[54] L. Holy, P. Picha, R. Lipka, and P. Brada. Software Engineering Projects Analysis using Interactive Multimodal Graph Explorer – IMiGEr. In *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: IVAPP*, pages 330–337, Setubal, Portugal, 2019. SciTePress.

[55] S. Jablonski, B. Volz, and S. Dornstauder. A meta modeling framework for domain specific process management. In *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*, pages 1011–1016, Piscataway, NJ, USA, 2008. IEEE.

[56] C. Jensen and W. Scacchi. Data mining for software process discovery in open source software development communities. In *Proc. Workshop on Mining Software Repositories*, pages 96–100, Stevenage, UK, 2004. IET.

[57] A. Jermakovics, A. Sillitti, and G. Succi. Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pages 24–31, New York, NY, USA, 2011. ACM.

[58] N. Jlaiel, K. Madhbouh, and M. B. Ahmed. A semantic approach for automatic structuring and analysis of software process patterns. *International Journal of Computer Applications*, 54(15):24–31, 2012.

[59] M. Joblin, S. Apel, C. Hunsen, and W. Mauerer. Classifying developers into core and peripheral: An empirical study on count and network metrics. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 164–174, Piscataway, NJ, USA, 2017. IEEE.

[60] M. Joblin, S. Apel, and W. Mauerer. Evolutionary trends of developer coordination: A network approach. *Empirical Software Engineering*, 22(4):2050–2094, 2017.

[61] M. Joblin, W. Mauerer, S. Apel, J. Siegmund, and D. Riehle. From developer networks to verified communities: a fine-grained approach. In *Proceedings of the 37th International Conference on Software Engineering – Volume 1*, pages 563–573, Piscataway, NJ, USA, 2015. IEEE Press.

[62] D. Johnston. Agile Anti-Patterns: A Systems Thinking Approach. Online, 2019. https://www.infoq.com/articles/agile-anti-patterns-systems-thinking/. Accessed June 20th, 2019.

[63] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, New York, NY, USA, 2014. ACM.

[64] M. Khaari and R. Ramsin. Process patterns for aspect-oriented software development. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pages 241–250, Piscataway, NJ, USA, 2010. IEEE.

[65] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, 84(4):559–572, 2011.

[66] B. Kitchenham and E. Mendes. Software productivity measurement using multiple size measures. *IEEE Transactions on Software Engineering*, 30(12):1023–1035, 2004.

[67] E. Kouroshfar, H. Y. Shahir, and R. Ramsin. Process patterns for component-based software development. In *International Symposium on Component-Based Software Engineering*, pages 54–68, New York, NY, USA, 2009. Springer.

[68] P. Kroll and P. Kruchten. *The Rational Unified Process made easy: A practitioner's guide to the RUP.* Addison-Wesley Professional, Boston, MA, USA, 2003.

[69] P. Kroll and B. MacIsaac. *Agility and Discipline Made Easy: Practices from OpenUP and RUP.* Pearson Education, London, UK, 2006.

[70] Y. Kuranuki and K. Hiranabe. Antipractices: Antipatterns for XP practices. In *Agile Development Conference, 2004*, pages 83–86, Piscataway, NJ, USA, 2004. IEEE.

[71] P. A. Laplante and C. J. Neill. *Antipatterns: Identification, Refactoring, and Management.* Auerbach Publications, New York, NY, USA, 2005.

[72] C. Y. Laporte, R. V. O'Connor, and G. Fanmuy. International systems and software engineering standards for very small entities. *CrossTalk – The Journal of Defense Software Engineering*, 26(3):28–33, 2013.

[73] Livejournal. Antipatterns. Online, 2008. https://thespleen.livejournal.com/109833.html. Accessed November 14th, 2018.

[74] N. Malik. Project Management AntiPattern – PMs who write specs. Online, 2006. https://blogs.msdn.microsoft.com/nickmalik/2006/01/03/project-management-antipattern-pms-who-write-specs/. Accessed November 14th, 2018.

[75] N. Malik. Project Management Antipattern 2: Pardon My Dust. Online, 2016. https://blogs.msdn.microsoft.com/nickmalik/2006/01/19/project-management-antipattern-2-pardon-my-dust/. Accessed November 14th, 2018.

[76] T. Martınez-Ruiz, F. Garcıa, M. Piattini, and J. Münch. Modelling software process variability: an empirical study. *IET software*, 5(2):172–187, 2011.

[77] A. Martini and J. Bosch. A multiple case study of continuous architecting in large agile companies: current gaps and the CAFFEA framework. In *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*, pages 1–10, Piscataway, NJ, USA, 2016. IEEE.

[78] A. Martini, L. Pareto, and J. Bosch. Towards introducing agile architecting in large companies: the CAFFEA framework. In *International Conference on Agile Software Development*, pages 218–223, New York, NY, USA, 2015. Springer.

[79] X.-x. Meng, Y.-s. Wang, L. Shi, and F.-j. Wang. A process pattern language for agile methods. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 374–381, Piscataway, NJ, USA, 2007. IEEE.

[80] I. Mitchell. Agile Patterns. Online, 2019. https://dzone.com/refcardz/agile-patterns. Accessed June 20th, 2019.

[81] A. Mockus and L. G. Votta. Identifying Reasons for Software Changes using Historic Databases. In *Proceedings of International Conference on Software Maintenance*, pages 120–130, Washington, DC, USA, 2000. IEEE Computer Society Press.

[82] S. Morasca and G. Russo. An empirical study of software productivity. In *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, pages 317–322, Piscataway, NJ, USA, 2001. IEEE.

[83] C. J. Neill. Effective Teams | Management Antipatterns – how NOT to manage teams. Online, 2018. http://www.personal.psu.edu/cjn6/Personal/Effective%20Teams.htm. Accessed November 14th, 2018.

[84] D. Nicolette. *Software development metrics*. Manning Publications, Shelter Island, NY, USA, 2015.

[85] R. Oberhauser. Leveraging Semantic Web Computing for Context-Aware Software Engineering Environments. In *Semantic Web*. IntechOpen, London, UK, 2010.

[86] Object Management Group. Software & Systems Process Engineering Meta-Model Specification, 2008.

[87] Object Management Group. Business Process Model and Notation (BPMN), 2011.

[88] T. Ohno. *Toyota production system: beyond large-scale production*. CRC Press, Boca Raton, FL, USA, 1988.

[89] OSLC. Open Services for Lifecycle Collaboration. Online, 2018. https://open-services.net/. Accessed November 14th, 2018.

[90] F. Palma, N. Moha, and Y.-G. Guéhéneuc. Specification and detection of business process antipatterns. In *International Conference on E-Technologies*, pages 37–52, Cham, Switzerland, 2015. Springer International Publishing.

[91] F. Palomba, D. A. Tamburri, A. Serebrenik, A. Zaidman, F. A. Fontana, and R. Oliveto. How do community smells influence code smells? In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 240–241, New York, NY, USA, 2018. ACM.

[92] M. Perkusich, G. Soares, H. Almeida, and A. Perkusich. A procedure to detect problems of processes in software development projects

using Bayesian networks. *Expert Systems with Applications*, 42(1):437–450, 2015.

[93] K. Petersen, P. Roos, S. Nyström, and P. Runeson. Early identification of bottlenecks in very large scale system of systems software development. *Journal of software: Evolution and Process*, 26(12):1150–1171, 2014.

[94] P. Picha and P. Brada. Empirical Research in Software Engineering: A Literature Review. In *The Ninth International Conference on Software Engineering Advances*, volume 7, pages 209–214, Wilmington, DE, USA, 2014. IARIA.

[95] P. Pícha and P. Brada. ALM tool data usage in software process metamodeling. In *Software Engineering and Advanced Applications (SEAA), 2016 42th Euromicro Conference on*, pages 1–8, Piscataway, NJ, USA, 2016. IEEE.

[96] P. Picha and P. Brada. Software Process Anti-pattern Detection in Project Data. In *EuroPLoP '19: Proceedings of the 24th European Conference on Pattern Languages of Programs*, New York, NY, USA, 2019. ACM.

[97] P. Pícha, P. Brada, R. Ramsauer, and W. Mauerer. Towards Architect's Activity Detection through a Common Model for Project Pattern Analysis. In *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pages 175–178, Piscataway, NJ, USA, 2017. IEEE.

[98] R. Ramsauer, D. Lohmann, and W. Mauerer. Observing Custom Software Modifications: A Quantitative Approach of Tracking the Evolution of Patch Stacks. In *Proceedings of the 12th International Symposium on Open Collaboration*, page 4, New York, NY, USA, 2016. ACM.

[99] C. Raptopoulou, E. Berki, T. Poranen, I. Stamelos, and L. Aggelis. Management anti-patterns in finnish software industry. In *Proceedings of the SQM/INSPIRE 2012 Conference*, pages 173–187, Tampere, Finland, 2012. School of Information Sciences of the University of Tampere and the BCS.

[100] J. Roa, E. Reynares, M. L. Caliusco, and P. Villarreal. Towards Ontology-Based Anti-patterns for the Verification of Business Process Behavior. In *New Advances in Information Systems and Technologies*, pages 665–673. Springer, New York, NY, USA, 2016.

[101] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Washington, DC, USA, 1987. IEEE Computer Society Press.

[102] F. B. Ruy, R. de Almeida Falbo, M. P. Barcellos, and G. Guizzardi. An Ontological Analysis of the ISO/IEC 24744 Metamodel. In *Formal Ontology in Information Systems*, pages 330–343, Amsterdam, Netherlands, 2014. IOS Press.

[103] Scaled Agile, Inc. Scaled Agile Framework – SAFe for Lean Development. Online, 2018. https://www.scaledagileframework.com/. Accessed November 14th, 2018.

[104] D. Settas, S. Bibi, P. Sfetsos, I. Stamelos, and V. Gerogiannis. Using bayesian belief networks to model software project management antipatterns. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 117–124, Piscataway, NJ, USA, 2006. IEEE.

[105] D. Settas and I. Stamelos. Towards a dynamic ontology based software project management antipattern intelligent system. In *Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on*, volume 1, pages 186–193, Piscataway, NJ, USA, 2007. IEEE.

[106] D. Settas and I. Stamelos. Using Ontologies to Represent Software Project Management Antipatterns. In *Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2007)*, pages 604–609, Skokie, IL, USA, 2007. Knowledge Systems Institute Graduate School.

[107] D. L. Settas, G. Meditskos, I. G. Stamelos, and N. Bassiliades. SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies. *Expert Systems with Applications*, 38(6):7633–7646, 2011.

[108] P. Silva, A. M. Moreno, and L. Peters. Software Project Management: Learning from Our Mistakes. *IEEE Software*, 32(3):40–43, May–June 2015.

[109] P. Smiari, S. Bibi, and I. Stamelos. Knowledge acquisition during software development: Modeling with anti-patterns. In *Synergies Between Knowledge Engineering and Software Engineering*, pages 75–92. Springer, New York, NY, USA, 2018.

[110] Sourcemaking.com. AntiPatterns. Online, 2018. https://sourcemaking.com/antipatterns. Accessed November 14th, 2018.

[111] I. Stamelos. Software project management anti-patterns. *Journal of Systems and Software*, 83(1):52–59, 2010.

[112] J. V. Sutherland and K. Schwaber. The Scrum methodology. In *Business object design and implementation: OOPSLA workshop*, Londen, UK, 1995. Springer-Verlag London.

[113] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.*, pages 183–192, Piscataway, NJ, USA, 2003. IEEE.

[114] H. Takeuchi and I. Nonaka. The new new product development game. *Harvard business review*, 64(1):137–146, 1986.

[115] D. A. Tamburri, P. Lago, and H. van Vliet. Uncovering Latent Social Communities in Software Development. *IEEE Software*, 30(1):29–36, 2013.

[116] D. A. Tamburri, F. Palomba, A. Serebrenik, and A. Zaidman. Discovering Community Patterns in Open-Source: A Systematic Approach and Its Evaluation. *Empirical Software Engineering*, 23:1–49, 2018.

[117] A. H. M. Ter Hofstede, C. Ouyang, M. La Rosa, L. Song, J. Wang, and A. Polyvyanyy. APQL: A process-model query language. In *Asia-Pacific Conference on Business Process Management*, pages 23–38, Cham, Switzerland, 2013. Springer International Publishing.

[118] The LeSS Company B.V. Overview – Large Scale Scrum (LeSS). Online, 2018. https://less.works/. Accessed November 14th, 2018.

[119] The Linux Foundation. Home – CHAOSS. Online, 2018. https://chaoss.community/. Accessed November 14th, 2018.

[120] W. M. P. van der Aalst. What makes a good process model? *Software & Systems Modeling*, 11(4):557–569, 2012.

[121] W. Wang, J. Yang, and P. S. Yu. Meta-patterns: Revealing hidden periodic patterns. In *Proceedings 2001 IEEE International Conference on Data Mining*, pages 550–557, Piscataway, NJ, USA, 2001. IEEE.

[122] Wikipedia. Anti-Pattern. Online, 2018. https://en.wikipedia.org/wiki/Anti-pattern. Accessed November 14th, 2018.

[123] P. Wisse. *Metapattern: context and time in information models.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[124] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, New York, NY, USA, 2004. ACM.

[125] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

# List of Abbreviations

**AGQM** Agile GQM. 36

**ALM** Application Lifecycle Management. 5, 7, 8, 25, 26, 29–32, 41, 43, 45, 51, 53, 55–61, 63, 64, 66–68, 70–76, 79, 80, 82, 91–96

**AOSD** Aspect-Oriented Software Development. 51

**API** Application Programming Interface. 34, 38, 46, 74, 75

**APPL** Agile PPL. 50

**APQL** A Process-model Query Language. 50

**BDTEX** Bayesian Detection Expert. 52

**BN** Bayesian (Belief) Network. 41, 51, 52, 88

**BPMN** Business Process Model and Notation. 44, 51, 52, 59, 89

**CAPDL** Control-flow Anti-Pattern Description Language. 51

**CBSE** Component Based Software Engineering. 51

**CHAOSS** Community Health Analytics Open Source Software project. 31, 32, 57

**CLI** command line interface. 57

**CLM** Collaborative Lifecycle Management. 30

**CMMI** Capability Maturity Model Integration. 50

**CoSEEEK** Context-aware Software Engineering Environment Event-driven frameworK. 35–37, 50, 60, 64, 77, 114

**CVS** Concurrent Version System. 26, 33, 34, 39

**DAD** Disciplined Agile Delivery. 23

**DEEP** Detailed appropriately, Emergent, Estimated, and Prioritized. 125

**eSPEM** enactable SPEM. 43

**ETL** Extract-Transform-Load. 33, 56, 75

**EUP** Enterprise Unified Process. 17

**GQM** Goal-Question-Metric. 51–53, 88

**GUI** graphical user interface. 33, 57, 64, 74, 75, 89, 90, 95, 114

**IMiGEr** Interactive Multimodal Graph Explorer. 91, 115

**JIT** just-in-time. 22, 23

**JSON** JavaScript Object Notation. 31, 51, 74, 91

**LeSS** Large Scale Scrum. 20

**LOC** lines of code. 33

**LSD** Lean Software Development. 22

**MSR** Mining Software Repositories. 8, 45, 73, 75

**NLP** Natural Language Processing. 89

**OSLC** Open Services for Lifecycle Collaboration. 43, 59, 60

**OSS** Open-Source Software. 6, 27, 29, 31–35, 37–39, 45, 53, 73–75, 93, 94

**OTH** Osterbayerische Technische Hochschule. 35, 94

**OWL** Web Ontology Language. 41

**PaStA** Patch Stack Analysis. 35, 53, 67, 73, 77, 92

**PM** Project Management. 5, 7, 8, 13, 27, 41, 43, 46, 52, 57, 76, 77, 80, 95, 96

**PMMM** Process Meta Meta Model. 44

**PPL** Pattern Process Language. 50

**PROMAISE** The Software Project Management Antipattern Intelligent System. 41, 50, 77

**RDBMS** relational database management system. 80

**REST** representational state transfer. 46, 74

**RMC** Rational Method Composer. 42

**ROSE** Reengineering of Software Evolution. 45

**RQ** research question. 7, 8, 57, 77, 89, 91, 96

**RTC** Rational Team Concert. 27, 60

**RUP** Rational Unified Process. 16, 17, 23, 61, 78

**SAFe** Scaled Agile Framework. 20

**SE** Software Engineering. 6, 10, 13, 20, 36, 41, 55, 60, 65, 67, 75, 94–96

**SEMDM** Software Engineering Metamodel for Development Methodologies. 43, 44

**SEWL** Software Engineering Workflow Language. 35, 50

**SOAP** Simple Object Access Protocol. 39

**SPADe** Software Process Anti-patterns Detector. 55, 56, 58–61, 65, 72, 74, 79, 82, 89–91, 93, 95, 96, 114, 115

**SPARSE** Symptom-based Antipattern Retrieval Knowledge based System Using Semantic Web Technologies. 41, 50, 77

**SPEM** Software & Systems Process Engineering Meta-Model. 42–44, 51, 58, 61

**SPI** Software Process Improvement. 7, 8, 35, 52, 57

**SQL** Structured Query Language. 33, 35, 80, 81, 89

**SVN** Subversion. 7, 26, 32, 60, 73, 75

**UI** user interface. 46

**UML** Unified Modeling Language. 12, 39, 43, 51

**UP** Unified Process. 16, 17, 23

**VCS** Version Control System. 7, 26–29, 33, 35, 36, 45, 60–62, 64, 72, 73, 125

**vSPEM** variability SPEM. 43

| | |
|---|---|
| **XML** | Extensible Markup Language. 44, 51, 74, 89 |
| **XP** | eXtreme Programming. 20, 22 |
| **XSD** | XML Schema Definition. 89 |
| **YOSHI** | Yielding Open-Source Health Information. 37, 38, 73, 114 |

# List of Figures

# List of Tables

# Appendices

# Appendix A

# Project Management Anti-patterns from Literature

Table A.1: Project management anti-patterns gathered from literature

| **Anti-pattern Name** | **Also Known As** | **Sources** |
|---|---|---|
| Absentee Manager | - | [71, 73, 108] |
| An Athena | - | [27] |
| Analysis Paralysis | Process Mismatch, Waterfall | [9, 17, 27, 99, 110, 122] |
| Anybody Syndrome | - | [70] |
| Appointed Team | - | [27, 108] |
| Architects Don't Code | - | [27] |
| Architects Play Golf | - | [28] |
| Band Aid | Buff And Shine A Rusty Car, Cosmetic Surgery, The Quick Fix | [27] |
| Bicycle Shed | - | [122] |
| Big Requirements Documentation | - | [33, 34] |
| Black-Cloud | - | [116] |
| Blame Storming | - | [27] |
| Blamer | - | [4] |
| Bleeding Edge | - | [122] |
| Blowhard Jamboree | - | [9, 17, 27, 99, 110] |
| Bottleneck | Radio-silence | [116] |
| Broken Iron Triangle | - | [3] |
| Brooks' Law | - | [18, 71, 99, 108, 122] |
| Brownie's Works | - | [70] |

| | | |
|---|---|---|
| Business As Usual | No Sprint Retrospective | [33] |
| Bystander Apathy | - | [122] |
| Cage Match Negotiator | - | [73] |
| Car Park Syndrome | - | [28] |
| Carbon Copy His Manager | - | [27] |
| Cargo Cult | - | [28] |
| Cart Before The Horse | - | [122] |
| Cash Cow | - | [122] |
| Change Prevention Process | - | [5] |
| Confusion Of Objectives | - | [27] |
| Copy And Paste Programming | - | [122] |
| Corncob | Corporate Shark, Loose Cannon, Third-World Information Systems Troubles | [9, 17, 27, 99, 110] |
| Creation Dependence | - | [49] |
| Cryptocracy | - | [28] |
| Customer Caused Disruption | - | [33, 34] |
| Customer Product Owner | - | [33, 34] |
| Death By Planning | Detailitis Plan | [9, 17, 27, 99, 108, 110] |
| Death March | - | [17, 122] |
| Decision By Arithmetic | Management By Numbers, Management By Objectives | [27, 122] |
| Dependency Hell | - | [122] |
| Design By Committee | - | [122] |
| Discordant Reward Mechanisms | - | [27] |
| DLL Hell | - | [122] |
| Doppelganger | - | [71, 73] |
| Dry Waterhole | - | [27, 108] |
| Egalitarian Compensation | - | [27] |
| Email Is Dangerous | Email Flaming | [9, 17, 27, 99, 110] |

| | | |
|---|---|---|
| Emperor's New Clothes | - | [27, 71, 99] |
| Empire Building | - | [27] |
| Escalation Of Commitment | - | [122] |
| Every Fool Their Own Tool | - | [122] |
| Extension Conflict | - | [122] |
| False Economy | - | [28] |
| False Surrogate Endpoint | - | [27] |
| Fear Of Success | - | [9, 17, 99] |
| Feature Creep | Requirement Creep, Scope Creep | [110, 122] |
| Fire Drill | - | [17, 108, 110] |
| Fruitless Hoops | - | [71, 73] |
| Fungible Project Manager | - | [27] |
| Fungible Teams | - | [27] |
| Geographically Distributed Development | - | [28] |
| Give Me Estimates Now | - | [27] |
| Glass Case Plan | - | [108] |
| Glass Wall | - | [27] |
| Golden Child | - | [73] |
| Groupthink | - | [122] |
| Guilding The Lily | - | [99] |
| Half Done Is Enough | - | [27] |
| Headless Chicken | - | [71, 73] |
| Heir Apparent | - | [27] |
| Hero Culture | Dragon Slaying | [27] |
| Hidden Requirements | - | [27, 99] |
| Hours In Progress Monitoring | - | [33] |
| Hypocritical Preacher | - | [4] |
| Idiot Proof Process | - | [27] |
| If It Is Working Don't Change | If It Ain't Broke Don't Fix It, You Aren't Gonna Need It | [27] |
| Improbability Factor | - | [122] |

| | | |
|---|---|---|
| Inappropriate Technical Objective | - | [27] |
| Indifferent Specialist | - | [4] |
| Inflexible Plan | - | [18, 108] |
| Intellectual Violence | - | [17, 110] |
| Invented Here | - | [122] |
| Invisible Progress | - | [33, 34] |
| Irrational Management | - | [9, 17, 71, 108, 110] |
| It's Not Rocket Science | - | [27] |
| JAR Hell | - | [122] |
| Leader Not Manager | - | [71, 73, 83, 108] |
| Leading Request | Paved With Good Intentions, Wild Goose Chase | [27] |
| List Dictator | - | [4] |
| Lone-Wolf | - | [116] |
| Long Or Non-Existent Feedback Loops | - | [33] |
| Manager Controls Process | - | [27] |
| Manager Not Leader | - | [71, 73, 83] |
| Managerial Cloning | - | [71, 73] |
| Metric Abuse | Bad Management By Metrics, Metric Madness | [71, 73, 111] |
| Micromanagement | - | [18, 108, 122] |
| Mind Reader | - | [4] |
| Moral Hazard | - | [122] |
| Mr. Nice Guy | - | [71, 73] |
| Mushroom Management | - | [27, 71, 108, 122] |
| Myopic Delivery | - | [18, 108] |
| Net Negative Producing Programmer | - | [27] |
| Ninety-Ninety Rule | - | [122] |
| Non-Creative Intelligence | - | [10, 111] |
| Not Invented Here | Reinventing The (Square) Wheel | [122] |
| Online Backstabber | - | [4] |

| | | |
|---|---|---|
| Organisational Silo | - | [116] |
| Ostrich | - | [73] |
| Overengineering | - | [122] |
| Pairing Prison | - | [70] |
| Pardon My Dust | - | [75] |
| Peter Principle | - | [122] |
| Plate Spinning | - | [71] |
| Plug Compatible Interchangeable Engineers | - | [27] |
| PMs Who Write Specs | - | [74] |
| Premature Optimization | - | [122] |
| Process Disintegration | - | [18, 108] |
| Product Owner Without Authority | - | [33, 34] |
| Programming By Permutation | Programming by Accident, Programming by Coincidence | [122] |
| Project Mismanagement | - | [9, 17, 108, 110] |
| Proletariat Hero | - | [71, 73, 108] |
| Rising Upstart | - | [71, 73, 108] |
| Road To Nowhere | - | [71, 108] |
| Scapegoat | - | [27] |
| Seagull Management | Hit and Run Management | [27, 73, 122] |
| Selling A Product You Can't Realize | - | [27] |
| Semi-Functional Teams | - | [33, 34] |
| Shaken But Not Stirred | - | [104] |
| Shoot The Messenger | Blame The Messenger, Kill The Messenger, Visibility Gets You Shot | [27] |
| Silver Bullet | All You Have Is a Hammer, Golden Hammer, One Trick Pony | [71, 108, 122] |
| Smoke And Mirrors | - | [9, 17, 99, 110, 122] |

| | | |
|---|---|---|
| Software Merger | - | [28] |
| Spammer | - | [4] |
| Specify Nothing | - | [27] |
| Spineless Executive | - | [71, 73] |
| Standing On The Shoulders Of Midgets | - | [27] |
| Stovepipe Or Silos | - | [122] |
| Subsequent Adjustment | - | [34, 49] |
| Tester Driven Development | - | [122] |
| Testing In Next Sprint | - | [33, 34] |
| The Brawl | - | [18, 108] |
| The Customer | - | [99] |
| The Customers Are Idiots | Users Are Idiots | [27] |
| The Domino Effect | - | [18, 108] |
| The Feud | Dueling Corncobs, Territorial Managers, Turf Wars | [9, 17, 110] |
| The Process Is The Deliverable | - | [27] |
| They Understood Me | - | [27] |
| Three Headed Knight | - | [71, 73] |
| Thrown Over The Wall | - | [9, 17, 27, 110] |
| Too Long Sprint | - | [33, 34] |
| Train The Trainer | - | [27, 99] |
| True Believer | Born Again Developer | [4] |
| Typecasting | - | [122] |
| Ultimate Weapon | - | [71, 73, 108] |
| Unjustified Criticizer | - | [4] |
| Unknown Poster | - | [4] |
| Unordered Product Backlog | - | [33, 34] |
| Untested But Finished | - | [28] |
| Varying Sprint Length | - | [33] |
| Vendor Lock-In | - | [122] |
| Vietnam War | - | [28] |
| Viewgraph Engineering | - | [9, 17, 27, 110] |

| | | |
|---|---|---|
| Warm Bodies | Deadwood, Mythical Man Month, Seat Warmers | [27, 71, 73, 99] |
| We Are Idiots | - | [27] |
| Work Estimates Given To Teams | - | [33, 34] |
| Yes Man | - | [73] |
| Yet Another Meeting Will Solve It | - | [27, 99] |
| Yet Another Programmer | - | [27] |
| Yet Another Thread Will Solve It | - | [28] |

# Appendix B

# Project Management Anti-patterns from Experience

Table B.1: Project management anti-patterns gathered from experience

| **Anti-pattern Name** | **Description** |
|---|---|
| Artifact M.I.A. | an important artifact that should already exist is nowhere to be found |
| Backlog Not DEEP | not using the DEEP[1] strategy (i.e., problem description, prioritization, decomposition and effort estimation) |
| Collective Procrastination | frantic effort after long period of stagnation and falling behind on the plan; general case of Fire Drill anti-pattern not tied to the change in pace coinciding with the stage switching from analysis to implementation |
| Done It? Tag It! | unable to recognize releases in VCS |
| Exhausting Laziness | commits small in numbers but large in scale |
| Insufficient Prioritization | not using prioritization in planning |
| Nine Pregnant Women | similar to Brooks' Law but focusing on the product quality, not project deadline (i.e., adding more developers in the middle of the project leads to worse quality of the outputs) |
| One Type Fits All | not signifying type of tasks or using just the default option for all |

[1]Detailed appropriately, Emergent, Estimated, and Prioritized (DEEP), a term coined by Roman Pichler and Mike Cohn [25].

| | |
|---|---|
| Overdue Work | tasks finished after their respective due dates |
| Over-/Underestimation | constant bad estimates of work |
| Poor SCM traceability | the relation between a task and resulting source code changes cannot be found |
| Poor Ticket Workflow | several statuses repeatedly skipped, even though specified in the workflow |
| Poor Work Decomposition | large tasks insufficiently broken down to easier to handle sub-tasks |
| Rogue Tasks | unplanned tasks |
| Team Keeps Hacking Away | team keeps making the same mistakes even after they are long apparent and should have been identified and mitigated |
| The Flash | not logging spent time resulting in tasks seemingly done in an instant |
| Ticket QA Potemkin | task completion is insufficiently verified by entity independent of its assignee |
| To Infinity And Beyond | tasks without a specified due date already in progress or finished |
| Unbalanced Workload | some team members overworked while others underutilized |
| Unfinished Business | tasks displaying two conflicting information about their completion |
| Uphill Battle | iteration backlog growing during its execution |
| What Are We Doing Again? | no trace of knowledge base for the project (e.g., wiki unused) |
| What Did I Do? | work without description, comments, documentation |
| Weekend Hackathons | Collective Procrastination, where the stagnation occurs during the week and effort on weekends |

# Appendix C

# Nine Pregnant Women Anti-pattern

Table C.1: Nine Pregnant Women – anti-pattern from experience

| | |
|---|---|
| **Name** | Nine Pregnant Women |
| **Also Known As** | Size Isn't Everything |
| **Summary** | An idea that adding more resources (developers) to the project will increase productivity or quality has the opposite effect. Leads to slower pace and drop in productivity and quality in general. Therefore failure to meet mid-project milestones, release criteria, etc. The name is derived from the colloquial expression "nine women can't make a baby in one month". |
| **Symptoms** | adding new people mid project |
| | the overall productivity stays the same (productivity per capita drops) or drops |
| | higher bug rate per developer, lower feature rate per developer, higher duplicate ratio |
| | higher communication overhead due to new people getting acquainted with the project |
| | underdelivering or overdue releases, milestones postponed |
| **Specific To** | - |
| **Related Anti-patters** | Brooks' Law – more specific case, adding new developers leads to overdue project |
| **Sources** | experience |

# Appendix D

# SQL Query for Collective Procrastination Anti-pattern

```
set @projectId = ?;
set @iterationId = ?;
set @daysLookahead = 3;
set @daysLookbehind = 6;
set @silenceSteepnes = 0.1;
set @cliffSteepnes = 2;
set @endDate = (
 select
  max(date_format(work_item.created, '%Y-%m-%d'))
 from
  ppicha.field_change,
  ppicha.work_item_change,
  ppicha.work_unit,
  ppicha.configuration_change,
  ppicha.work_item
 where
  field_change.name = 'status'
  and (field_change.newValue = 'Closed' or newValue = 'Invalid')
  and field_change.workItemChangeId = work_item_change.id
  and work_item_change.workItemId = work_unit.id
  and work_unit.iterationId = @iterationId
  and work_item_change.id = configuration_change.changeId
  and configuration_change.configurationId = work_item.id
);
set @startDate = (
 select
  min(date_format(work_item.created, '%Y-%m-%d'))
 from
```

```
  ppicha.work_unit,
  ppicha.work_item
 where
  work_unit.id = work_item.id
  and work_unit.iterationId = @iterationId
);
set @issuesCount = (
 select
  count(id)
 from
  ppicha.work_unit
 where
  work_unit.iterationId = @iterationId);
set @duration = (select datediff(@endDate, @startDate)+1);
set @startDateFormatted =
 date_format(@startDate, '%Y-%m-%d');
set @endDateFormatted = date_format(@endDate, '%Y-%m-%d');
set @dailyIdeal = (select @issuesCount/@duration);
set @silenceThreshold = @silenceSteepnes*@issuesCount;
set @cliffThreshold =
 @cliffSteepnes*@daysLookahead*@dailyIdeal;

select
 'closedOn',
 'lookbehind',
 'silenceThreshold',
 'lookahead',
 'cliffThreshold',
 'detected'
union
select
 results.closedOn,
 results.lookbehind,
 @silenceThreshold,
 results.lookahead,
 @cliffThreshold,
 if (
  results.dayIndex >= @daysLookbehind and
  results.lookbehind <= @silenceThreshold and
  results.lookahead >= @cliffThreshold, true, false
 ) detected
from (
 select
  @curRow := @curRow+1 as dayIndex,
  days.selected_date as closedOn,
  ifnull(dataPoints.lookbehind, 0) as lookbehind,
  ifnull(dataPoints.lookahead, 0) as lookahead
from (
  select
   *
  from (
   select
    adddate(
     @startDateFormatted, t3.i*1000+t2.i*100+t1.i*10+t0.i
    ) selected_date
   from
    (select 0 i union select 1 union select 2
     union select 3 union select 4 union select 5
     union select 6 union select 7 union select 8
     union select 9
    ) t0,
    (select 0 i union select 1 union select 2
     union select 3 union select 4 union select 5
     union select 6 union select 7 union select 8
     union select 9
    ) t1,
    (select 0 i union select 1 union select 2
     union select 3 union select 4 union select 5
     union select 6 union select 7 union select 8
     union select 9
    ) t2,
    (select 0 i union select 1 union select 2
     union select 3 union select 4 union select 5
     union select 6 union select 7 union select 8
     union select 9
    ) t3
  ) v
  where
   v.selected_date
    between @startDateFormatted and @endDateFormatted
) days
left join (
  select
   ifnull(lookbehind.dataPoint, lookahead.dataPoint)
    as dataPoint,
   ifnull(lookbehind.closedCount, 0) as lookbehind,
   ifnull(lookahead.closedCount, 0) as lookahead
  from (
select
 days.selected_date as dataPoint,
 count(distinct work_unit.id) as closedCount
from (
 select
  *
 from (
  select
   adddate(
    @startDateFormatted, t3.i*1000+t2.i*100+t1.i*10+t0.i
   ) selected_date
  from
   (select 0 i union select 1 union select 2
    union select 3 union select 4 union select 5
    union select 6 union select 7 union select 8
    union select 9
   ) t0,
   (select 0 i union select 1 union select 2
    union select 3 union select 4 union select 5
    union select 6 union select 7 union select 8
    union select 9
   ) t1,
   (select 0 i union select 1 union select 2
    union select 3 union select 4 union select 5
    union select 6 union select 7 union select 8
    union select 9
   ) t2,
   (select 0 i union select 1 union select 2
    union select 3 union select 4 union select 5
    union select 6 union select 7 union select 8
    union select 9
   ) t3
 ) v
 where
  v.selected_date
   between @startDateFormatted and @endDateFormatted
) days,
 ppicha.work_item,
 ppicha.configuration_change,
 ppicha.work_item_change,
 ppicha.work_unit,
 ppicha.field_change,
 ppicha.status,
 ppicha.project_instance,
    ppicha.project,
    ppicha.status_classification
   where
    date_format(work_item.created, '%Y-%m-%d')
     > days.selected_date
    and date_format(work_item.created, '%Y-%m-%d')
     <= adddate(days.selected_date,
      interval @daysLookahead day)
    and work_item.id = configuration_change.configurationId
    and configuration_change.changeId = work_item_change.id
    and work_item_change.workItemId = work_unit.id
    and work_unit.iterationId = @iterationId
    and work_item_change.id = field_change.workItemChangeId
    and field_change.name = 'status'
    and field_change.newValue = status.name
    and status.projectInstanceId = project_instance.id
    and project_instance.projectId = @projectId
    and status.classId = status_classification.id
    and status_classification.superClass = 'CLOSED'
   group by
    dataPoint
  ) lookahead
  left join (
   select
    days.selected_date as dataPoint,
    count(distinct work_unit.id) as closedCount
   from (
    select
     *
    from (
     select
      adddate(
       @startDateFormatted, t3.i*1000+t2.i*100+t1.i*10+t0.i
      ) selected_date
     from
      (select 0 i union select 1 union select 2
       union select 3 union select 4 union select 5
       union select 6 union select 7 union select 8
       union select 9
      ) t0,
      (select 0 i union select 1 union select 2
       union select 3 union select 4 union select 5
       union select 6 union select 7 union select 8
       union select 9
      ) t1,
      (select 0 i union select 1 union select 2
       union select 3 union select 4 union select 5
       union select 6 union select 7 union select 8
       union select 9) t2,
      (select 0 i union select 1 union select 2
       union select 3 union select 4 union select 5
       union select 6 union select 7 union select 8
       union select 9
      ) t3
    ) v
    where
     v.selected_date
      between @startDateFormatted and @endDateFormatted
   ) days,
    ppicha.work_item,
    ppicha.configuration_change,
    ppicha.work_item_change,
    ppicha.work_unit,
    ppicha.field_change,
    ppicha.status,
    ppicha.project_instance,
    ppicha.project,
    ppicha.status_classification
   where
    date_format(work_item.created, '%Y-%m-%d')
     <= days.selected_date
    and date_format(work_item.created, '%Y-%m-%d')
     > subdate(days.selected_date,
     interval @daysLookbehind day)
    and work_item.id = configuration_change.configurationId
    and configuration_change.changeId = work_item_change.id
    and work_item_change.workItemId = work_unit.id
    and work_unit.iterationId = @iterationId
    and work_item_change.id = field_change.workItemChangeId
    and field_change.name = 'status'
    and field_change.newValue = status.name
    and status.projectInstanceId = project_instance.id
    and project_instance.projectId = @projectId
    and status.classId = status_classification.id
    and status_classification.superClass = 'CLOSED'
   group by
    dataPoint
  ) lookbehind
```

```
 on lookahead.dataPoint = lookbehind.dataPoint
union
select
 ifnull(lookbehind.dataPoint, lookahead.dataPoint)
  as dataPoint,
 ifnull(lookbehind.closedCount, 0) as lookbehind,
 ifnull(lookahead.closedCount, 0) as lookahead
from (
 select
  days.selected_date as dataPoint,
  count(distinct work_unit.id) as closedCount
 from (
  select
   *
  from (
   select
    adddate(
     @startDateFormatted, t3.i*1000+t2.i*100+t1.i*10+t0.i
    ) selected_date
   from
    (select 0 i union select 1 union select 2
     union select 3 union select 4 union select 5
     union select 6 union select 7 union select 8
     union select 9
    ) t0,
    (select 0 i union select 1 union select 2
     union select 3 union select 4 union select 5
     union select 6 union select 7 union select 8
     union select 9
    ) t1,
    (select 0 i union select 1 union select 2
     union select 3 union select 4 union select 5
     union select 6 union select 7 union select 8
     union select 9
    ) t2,
    (select 0 i union select 1 union select 2
     union select 3 union select 4 union select 5
     union select 6 union select 7 union select 8
     union select 9
    ) t3
  ) v
  where
   v.selected_date
    between @startDateFormatted and @endDateFormatted
  ) days,
   ppicha.work_item,
   ppicha.configuration_change,
   ppicha.work_item_change,
   ppicha.work_unit,
   ppicha.field_change,
   ppicha.status,
   ppicha.project_instance,
   ppicha.project,
   ppicha.status_classification
  where
   date_format(work_item.created, '%Y-%m-%d')
    > days.selected_date
   and date_format(work_item.created, '%Y-%m-%d')
    <= adddate(days.selected_date,
    interval @daysLookahead day)
   and work_item.id = configuration_change.configurationId
   and configuration_change.changeId = work_item_change.id
   and work_item_change.workItemId = work_unit.id
   and work_unit.iterationId = @iterationId
   and work_item_change.id = field_change.workItemChangeId
   and field_change.name = 'status'
   and field_change.newValue = status.name
   and status.projectInstanceId = project_instance.id
   and project_instance.projectId = @projectId
   and status.classId = status_classification.id
   and status_classification.superClass = 'CLOSED'
  group by
   dataPoint
 ) lookahead
 right join (
  select
   days.selected_date as dataPoint,
   count(distinct work_unit.id) as closedCount
  from (
   select
    *
   from (
    select
     adddate(
      @startDateFormatted, t3.i*1000+t2.i*100+t1.i*10+t0.i
     ) selected_date
    from
     (select 0 i union select 1 union select 2
        union select 3 union select 4 union select 5
        union select 6 union select 7 union select 8
        union select 9
       ) t0,
       (select 0 i union select 1 union select 2
        union select 3 union select 4 union select 5
        union select 6 union select 7 union select 8
        union select 9
       ) t1,
       (select 0 i union select 1 union select 2
        union select 3 union select 4 union select 5
        union select 6 union select 7 union select 8
        union select 9
       ) t2,
       (select 0 i union select 1 union select 2
        union select 3 union select 4 union select 5
        union select 6 union select 7 union select 8
        union select 9
       ) t3
     ) v
     where
      v.selected_date
       between @startDateFormatted and @endDateFormatted
   ) days,
     ppicha.work_item,
     ppicha.configuration_change,
     ppicha.work_item_change,
     ppicha.work_unit,
     ppicha.field_change,
     ppicha.status,
     ppicha.project_instance,
     ppicha.project,
     ppicha.status_classification
   where
     date_format(work_item.created, '%Y-%m-%d')
       <= days.selected_date
     and date_format(work_item.created, '%Y-%m-%d')
       > subdate(days.selected_date,
       interval @daysLookbehind day)
     and work_item.id = configuration_change.configurationId
     and configuration_change.changeId = work_item_change.id
     and work_item_change.workItemId = work_unit.id
     and work_unit.iterationId = @iterationId
     and work_item_change.id = field_change.workItemChangeId
```

```
    and field_change.name = 'status'
    and field_change.newValue = status.name
    and status.projectInstanceId = project_instance.id
    and project_instance.projectId = @projectId
    and status.classId = status_classification.id
    and status_classification.superClass = 'CLOSED'
   group by
    dataPoint
  ) lookbehind
  on lookahead.dataPoint = lookbehind.dataPoint
 ) dataPoints
 on days.selected_date = dataPoints.dataPoint
 join
  (select @curRow := 0) r
 order by
  days.selected_date
) results
```