Supporting Multiple Stakeholders in Agile Development

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Nitish Patkar
aus Mumbai, Indien.

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Institut für Informatik
Universität Bern

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 29 March 2022

Der Dekan:
Prof. Dr. Zoltan Balogh

To my late father, who unconditionally accepted all my decisions.

Oscar Nierstrasz, working with you was an enriching experience. I will cherish this time forever.

Andrei Chis, without your enthusiasm, motivation, and support I wouldn't finish this thesis.

Everyone else, who put up with me.

Abstract

Agile software development practices require several stakeholders with different kinds of expertise to collaborate while specifying requirements, designing and modeling software, and verifying whether developers have implemented requirements correctly. We studied 112 requirements engineering (RE) tools from academia and the features of 13 actively maintained behavior-driven development (BDD) tools, which support various stakeholders in specifying and verifying the application behavior. Overall, we found that there is a growing tool specialization targeted towards a specific type of stakeholders. Particularly with BDD tools, we found no adequate support for non-technical stakeholders— they are required to use an integrated development environment (IDE)— which is not adapted to suit their expertise.

We argue that employing separate tools for requirements specification, modeling, implementation, and verification is counterproductive for agile development. Such an approach makes it difficult to manage associated artifacts and support rapid implementation and feedback loops.

To avoid dispersion of requirements and other software-related artifacts among separate tools, establish traceability between requirements and the application source code, and streamline a collaborative software development workflow, we propose to adapt an IDE as an agile development platform. With our approach, we provide in-IDE graphical interfaces to support non-technical stakeholders in creating and maintaining requirements concurrently with the implementation. With such graphical interfaces, we also guide non-technical stakeholders through the object-oriented design process and support them in verifying the modeled behavior. This approach has two advantages: (i) compared with employing separate tools, creating and maintaining requirements directly within a development platform eliminates the necessity to recover trace links, and (ii) various natively created artifacts can be composed into stakeholder-specific interactive live in-IDE documentation. These advantages have a direct impact on how various stakeholders collaborate with each other, and allow for rapid feedback, which is much desired in agile practices. We exemplify our approach using the Glamorous Toolkit IDE. Moreover, the discussed building blocks can be implemented in any IDE with a rich-enough graphical engine and reflective capabilities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It has been about twenty years since the agile manifesto was published in 2001. Although the word “agile” is quite popular and mainstream since then, development strategies that promoted iterative and incremental development have existed even longer since the early 1970s. After decades of development, agile practices are now widely adopted globally [88, 47, 108, 64]. However, as we later discuss in chapter 2, our analysis of the state of the art of various software engineering (SE) activities shows that some of them are ill-integrated and lack support for agile development. There are essentially four software engineering activities [121]:

1. *Specification*: wherein the functionality of the software and constraints on its operation are defined,

2. *Design and implementation*: wherein software to meet the specification is built,

3. *Validation*: wherein software is validated to ensure it does what the customer expects, and

4. *Evolution*: wherein software evolves to meet changing customer needs.

Development strategies, such as DevOps, promote using toolchains for faster feedback loops to support agility. Figure 11 shows a typical DevOps cycle and various phases in it. These toolchains often automate coding, building, testing, packaging, releasing, configuring, and monitoring of software. However, to the best of our knowledge, activities that demand direct collaboration among various stakeholders, such as specification and validation, lack any direct integration in such toolchains. Yet, it does not mean that there is a lack of tools to support specification and validation of requirements, or more generally, RE. Our analysis of the state of the art shows that there are, in fact, numerous tools that support various phases of RE. In our opinion, building dedicated tools to be eventually integrated

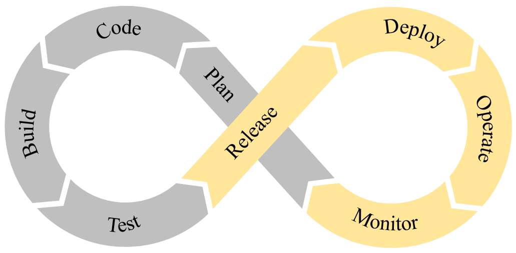


Figure 11: DevOps cycle

into toolchains is rather counterproductive for agile development as it requires significant effort to synchronize and trace several software-related artifacts produced in these tools. Besides, various stakeholders are forced to work in their own tool bubbles, blurring the overall product vision.

To support agile (*i.e.*, iterative, incremental, collaborative, and rapid) development, we believe that one platform should be used to create and maintain all software-related artifacts— requirements, design, models, and the source code. The recent advancements in development environments made us revisit various existing research ideas, such as the naked objects pattern and low-code development platforms (LCDPs), which present one way or the other to engage various types of stakeholders in the development process. We wonder whether these ideas can be adapted in integrated development environments (IDEs) to support agile development. The moldable development approach has already demonstrated how various in-IDE development-related tools (*e.g.*, debugger and coder) can be adapted to the underlying business domain to make software *explainable* [35]. The Glamorous Toolkit IDE (henceforth, called the Glamorous toolkit) is such a moldable development environment, which supports, through its numerous components, agile development [3]. The Glamorous toolkit presents us with an opportunity to develop a proof of concept to demonstrate how we turn an IDE into a collaborative agile development platform, especially for the specification, validation, and management of requirements.

1.1 Problem statement

Hypothesis. We propose to transform an IDE into a collaborative software engineering platform for multiple stakeholders by providing graphical user interfaces (GUIs) that enable the creation and manipulation of business objects.[1] Such a graphically enhanced IDE can support non-technical

[1]Business objects represent a thing active in the business domain, including at least its business name and definition, attributes, behavior, relationships, and constraints. Through business objects, managers and users can understand each other by using familiar concepts and creating a common model for interactions [52].

stakeholders in performing software engineering tasks, particularly, requirements specification, modeling, and management, concurrently with the software developers writing source code. Consequently, we answer the following research question:

How can we adapt an IDE for technical and non-technical stakeholders to create, validate, and overall manage requirements concurrently with the implementation?

1.2 Contributions

There are three main contributions of this doctoral thesis:

1.2.1 A survey of state of the art collaborative approaches

There exist numerous tools to support any one of the many RE activities. Likewise, numerous tools automate the BDD process, supporting domain experts in describing and verifying application behavior. We surveyed the state of the art tool landscape for RE and BDD. We discuss its limitations for being effective in an agile environment. We also present the results of an analysis of open-source projects and discuss the peculiarities of the current adoption of BDD in practice.

1.2.2 A survey of the 62 software-related artifacts

Numerous artifacts exists that facilitate work activities relating to a software project. These include, specifying requirements, designing, or modeling. We provide an overview of the 62 software-related artifacts identified from the existing literature. We found that a significant number of artifacts (*i.e.*, 54 out of 62) are introduced in the early SDLC phases but, are in fact, consumed in the later SDLC phases, presumably by technical stakeholders. Furthermore, most artifacts (39 out of 62) have a mixed format, meaning they need IDE capabilities to render both textual and graphical characteristics.

1.2.3 A proposal of a novel approach for agile specification and verification

We describe an approach, which we call *citizen requirements*, to streamline a collaborative development workflow. As a part of the citizen requirements approach: (i) we discuss how various software-related artifacts can be created natively in an IDE to specify and maintain requirements, and (ii) we discuss BDD as a representative behavior specification and verification strategy, and propose a novel in-IDE collaborative workflow support for it. We discuss the building blocks that can be implemented in any IDE with rich enough support for visualization and reflection. We exemplify our approach in the Glamorous toolkit. We show how distinct stakeholders can be supported in specifying requirements at varying levels of detail and verifying those with representations specific to a particular application domain, natively in the Glamorous toolkit. We evaluate our approach by conducting a survey to gather feedback on the approach's perceived usefulness in practice. Finally, we discuss two main implications of our research proposal. There are two main implications of our approach. First, it is a step towards maintaining live project documentation directly in an IDE. Second, the same approach can be adapted to streamline an agile modeling workflow. Overall, our approach could enable the integration of specification and design phases into a DevOps cycle.

1.3 Outline

This doctoral thesis is structured as follows:

- In chapter 2,
  - we present the results of the analysis of state of the art RE and BDD tools. We present the results of a systematic literature review (SLR) of the current RE tools.
  - We present the results of an analysis of open-source projects hosted on GitHub to characterize the BDD usage in practice. We considered BDD as the current support for BDD is exclusively IDE-based, which is the main focus of this doctoral thesis.

    ✍ The results of this study have been accepted for publication at the *29th IEEE International Conference on Software Analysis, Evolution and Re-engineering (SANER)* [32].
  - We also present the results of the feature analysis of 13 popular BDD tools.

    ✍ The results of this study have been published in the *Proceedings of the 25th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings* [97].

  Overall, we summarize the main limitations of the current RE and BDD tools in support of agility.

- In chapter 3,
  - we briefly discuss three existing research ideas and a concept, which we adapt in building our proof of concept. We describe

the underlying idea of citizen requirements and three building blocks, which can be implemented in any IDE. Finally, we briefly discuss the implications of our proposal.

✍ The results of this study have been published in the *Proceedings of the 19th Belgium-Netherlands software evolution workshop (Benevol 2020)* [96].

- In chapter 4,
  - we describe a collaborative workflow for creating and maintaining software-related artifacts in an IDE. We exemplify our idea by presenting three such artifacts. We also present the results of a pilot survey we conducted to evaluate the perceived usefulness of such a workflow.

    ✍ The results of this study have been accepted for publication at the *29th IEEE International Conference on Software Analysis, Evolution and Re-engineering (SANER)* [98].
- In chapter 5,
  - we describe a collaborative workflow for BDD, wherein non-technical stakeholders compose behavior scenarios using GUIs and verify the behavior using a domain-specific representation of the involved business objects.

    ✍ The results of this study have been published in the *Proceedings of the 25th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings* [97].
- In chapter 6,
  - we discuss two of the main implications of the citizen requirements approach: on project documentation and domain modeling.

    ✍ The live documentation part have been accepted for publication at the *29th IEEE International Conference on Software Analysis, Evolution and Re-engineering (SANER)* [98].
- Finally, chapter 7 concludes this doctoral thesis.
- In Appendix A, we provide further supporting details, such as the methodologies we followed for the several analysis projects and the corresponding threats to the validity, for those who are interested in exploring those details.

Chapter 2

State of the art

Before we present our vision of an agile development platform, it is important to explore the existing methods of specifying and verifying requirements and point out their limitations. In this chapter, we present the results of our analysis of the state of art:

- *RE tools*: in section 2.1, we present the details of the 112 RE tools proposed by researchers in recent years. We discuss their characteristics and point out that there is a growing tool specialization for various RE activities.
- *Software artifacts*: in section 2.2, we present the details of 62 software-related artifacts used to accomplish one of the several software engineering tasks. We discuss why it makes sense to create and maintain them in a single platform.
- *BDD tools*: in section 2.3, we present the details of 13 popular and well-maintained BDD tools and show that their features are inadequate to engage non-technical stakeholders in the BDD process.

2.1 RE tools

RE is a well-established discipline within software engineering. For more than 40 years, the RE community has been active in identifying and reacting to the challenges pertaining to, for example, communication or specification of requirements [25, 93, 33, 43]. In recent years, RE has been challenged by several developments within SE, such as the adoption of agile methodologies for software development. We did a systematic literature review (SLR) to provide an overview of the "tool-landscape" in RE research and report on how they support agile development. In this SLR, we examine literature published in recent years, *i.e.*, 2015-2019, in well-known SE venues and journals.

We closely followed Keele's comprehensive guidelines [74], which make it less likely that the results of the SLR will be biased. This method offers a means to evaluate and interpret research relevant to a topic of interest by evidence that is robust and transferable. In a nutshell,

- We used as our data sources the proceedings of the top nine SE conferences and all issues from of top SE journals for the years 2015-2019.
- After applying inclusion and exclusion criteria, we included 112 (55%) relevant publications for detailed analysis.

We have included the details of the methodology we followed in section A.1. Additionally, in section A.2, we present an overview of the identified RE tools, while section A.3 summarizes the threats to the validity.

2.1.1 Results

Publication trends

We first summarize the distribution characteristics of the publications, specifically, according to the year of publication, citations obtained in each track, venue, and corresponding tracks. Figure 21 shows the number of publications presented from different venues per year. The x-axis represents the venues and the y-axis denotes the number of found publications split by the different years. It is evident that the overall number of RE tools is relatively increasing during 2015-2019, except for 2019. It is

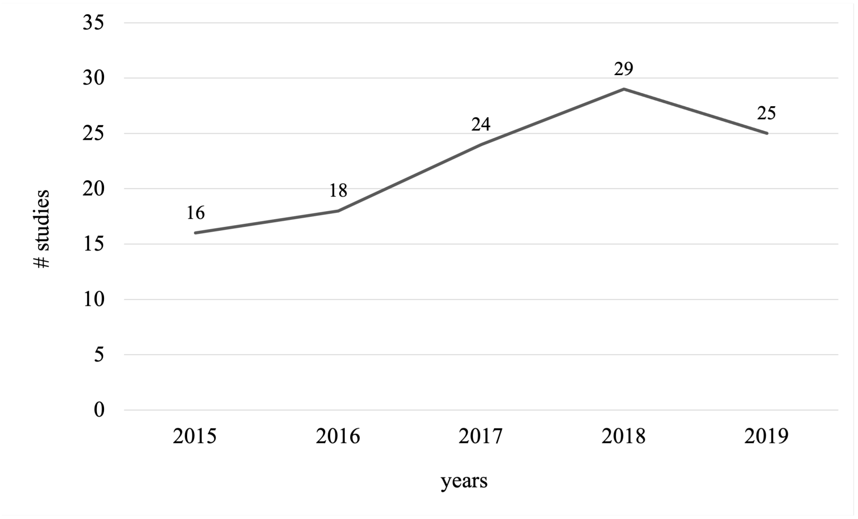


Figure 21: Growing numbers of RE tools

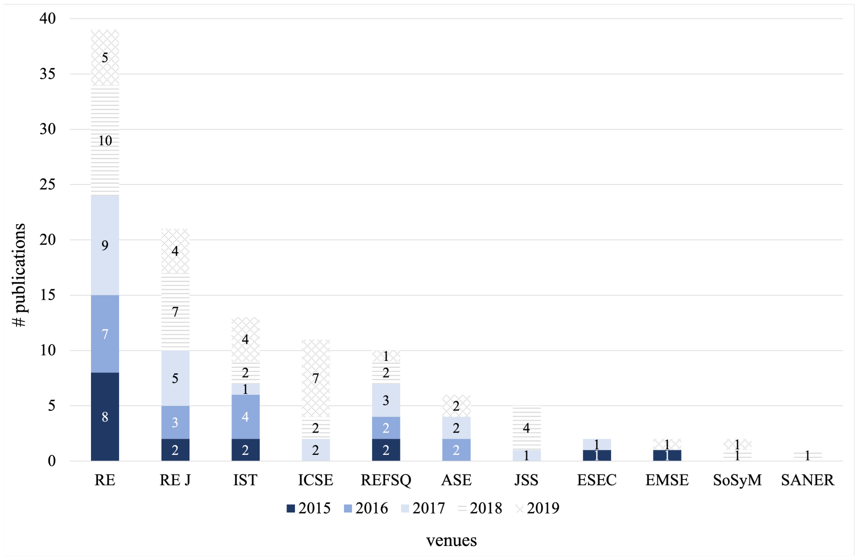


Figure 22: Topic awareness in different venues

evident from Figure 22 that, except for the dedicated RE venue, *i.e.*, the *RE* conference and the *RE Journal*, other top venues have rather modest representation of RE-related publications.

RE tool correspondence

Figure 23 shows the RE tool correspondence in various categories, such as whether they have been evaluated or are open-source, for each track. Note that the same tool might appear multiple times for different characteristics, *i.e.*, a prototype tool can be open source as well. The x-axis denotes the number of publications, whereas the y-axis correlates the affinity of the publications to the characteristics across four publication types, *i.e.*, three conference tracks and journal publications. The grey area is used to illustrate the total number of included publications in each publication type. The grey area indicates the whole to part relationship. As expected, a significant number of tools (a total of 37) are published in a tool demo track. Moreover, the percentage of evaluated studies is significantly higher in research tracks and journals, *i.e.*, 83% of studies are evaluated in research and 95% in journals. This indicates a more thorough review process for journals. We can further see that journal publications (38%) are the most prevalent in our data set, followed by tool demo (33%) and research papers (26%).

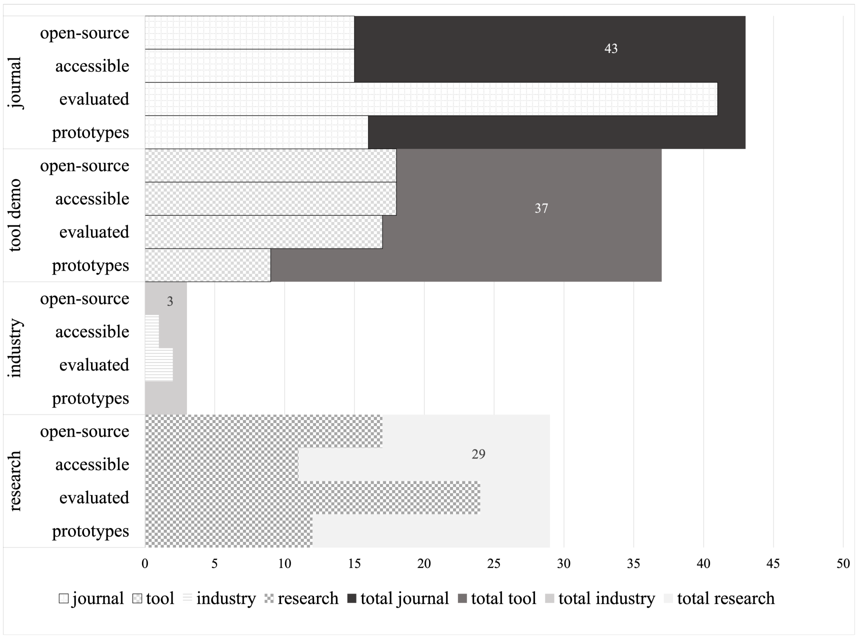


Figure 23: RE tool affinity for different tracks

Supported RE activities

The distribution of the tools according to the supported RE activity can be seen in Figure 24. The x-axis reports the different RE activities, whereas the y-axis shows the corresponding number of publications. The studies in which the supported RE activity is clearly mentioned by the authors are marked *explicit*, whereas for the remaining studies we had to read the paper to understand the focus, and consequently these are marked as *implicit*. We validated the assignment to mitigate any selection bias. A typical example is study [S42], which analyses user reviews received on application hosting platforms, such as Google Play, to specify features that we exclusively assigned to "specification." Clearly, the activities "specification" and "analysis" are the most prevalent, whereas "elicitation," "management," and "validation" are supported by fewer than 20% of all tools. We only see negligible differences for management in terms of implicit and explicit tools, simply because we classified prioritization and traceability tools as management tools. Similarly, we see negligible differences for specification in terms of implicit and explicit tools, simply because we classified modeling tools as specification tools. 18 authors claimed that their tools can support multiple activities. Still, a significant number of tools (*i.e.*, 89 of 112) only support a single RE activity, regardless of whether it an implicit or explicit assignment.

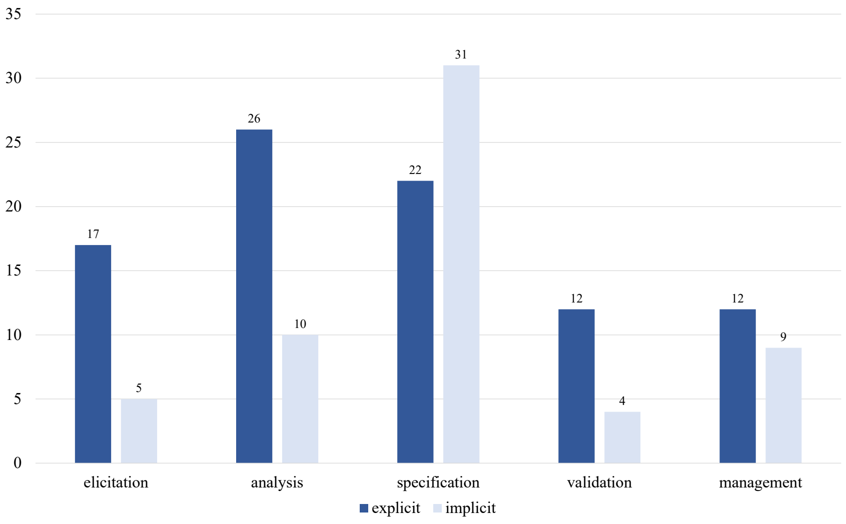


Figure 24: Tool support for RE activities

Overall, all RE activities are moderately supported by the proposed tools, specification being supported the most, whereas management being supported the least (kindly refer to Table A4).

Ambreen *et al.* complained that RE solutions mainly address the middle and late stages of the software development life cycle. Additionally, they pointed out that most research projects focus on a single RE problem [25, 33]. They found that the existing methods are not adequate for explicitly capturing and representing "business and organizational knowledge." Consequently, they expressed a need to integrate RE tools into a coherent requirements process. Similarly, the empirical work in the area of requirements validation and verification is very limited and shows a decreasing trend [13]. Despite a decade of research activities, previous observations are still valid.

Unlike the view expressed by Alves *et al.*, we think that supporting all RE activities does not necessarily require cooperation from several researchers to work jointly, but, instead, needs a different approach [11]. We believe that proposing distinct tools for distinct tasks might create an unnecessary gap between stakeholders, and lead to traceability issues, which is evident from the challenges reported in a review of 600 research articles on the software product lines (SPL) [34].

Target audience

Table 21 summarizes how many times target audience was reported by the authors. As we can see, most of the tools are crafted for requirements engineers, developers, and other non-technical stakeholders, while 33 tools (29%) targeted multiple roles.

Table 21: Count of the target audience

Role	Count
Requirements engineer	33
Developer	16
Non-technical stakeholder	10
end users	3
Business analyst	2
Requirements analyst	2
Manager	2
Product manager	2
Software architect	2
Domain expert	2
Designer	1
Minute taker	1
User story writer	1
Tester	1
Domain architect	1
Data scientist	1
Test manager	1
Test engineer	1
Software engineer	1
Project manager	1

RE tools are not restricted to support requirements engineers alone; they also aid other stakeholders, such as developers (kindly refer to Table 21).

Software projects are a joint endeavor. Several stakeholders, including technical and non-technical ones, have distinct responsibilities to make the project a success. Although requirements engineers seem to be the central audience in most cases, developers and other non-technical stakeholders were also mentioned several times. If we compare the target audience with the supported RE activity, we see that several tools now encourage developers and non-technical stakeholders to participate in requirements specification and management. It indicates a decisive shift towards involving various stakeholders in RE activities.

Prototypes

The comparison between prototypes, *i.e.*, the tools claimed by authors to be prototypes, and other tools can be seen in Figure 25. The x-axis denotes the number of tools, whereas the y-axis compares prototypes against other tools in various categories, such as whether they have been evaluated or are open-source. Note that the same tool might appear with multiple characteristics, *i.e.*, an open source tool can be evaluated as well. The grey area is used to illustrate the total number of included publications in each category. The grey area helps us to visualize the whole to part relationship. We found that a significant proportion of the proposed tools, *i.e.*, about

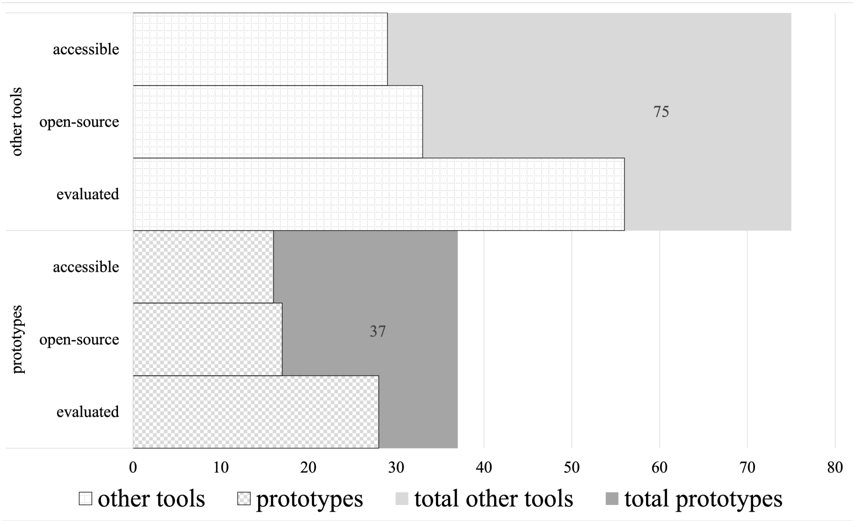


Figure 25: Prevalence of tool prototypes

33%, were considered to be prototypes by the respective authors. A web page is maintained for 16 of these prototype tools. Only four of these webpages contain URLs to the respective source code repository, making the rest hard to discover.

A large number of RE tools (33%) are research prototypes that are not readily available to a larger community (kindly refer to Table A4 and Figure 25).

Unlike industry projects, research projects focus on investigating new ideas or facing promising research directions. Typical outcomes of research projects are concepts, feasibility studies of novel approaches, and prototype applications for demonstration purposes. Such research prototypes undergo several maturity levels, *i.e.*, vision, concept, research prototype, quality-assured prototype, industry product, until they are adapted to an industrial context [144].

We speculate that the word "prototype" was perhaps used to indicate that the tool in its current state only supports a limited number of features and specific functionality. They were presumably proposed by the the author as a throwaway proof of concept or to be used by a community close to the author to obtain valuable feedback, and to provoke discussions on new ideas. Therefore, a large number of these research prototypes do not have an online presence.

Collaborative tools

Only ten tools were claimed to be collaborative. These tools were mainly intended to be used by a mixed audience, for example requirements engineers, end users, or developers. We further found that three of these ten tools are web applications, one is a desktop app, and one is a mobile application. The diversity of explicitly designated types of tools led us to the question "how do authors characterize support for collaboration in RE tools?" The corresponding details were largely missing from all studies. We contacted authors to learn what they understand by "collaboration." In essence, we learned that the term has not been used consistently. For the majority of tools, the authors used the term to emphasize that the tool serves multiple roles, such as developers and requirements engineers, and not necessarily that it allows simultaneous operations on the same model such as in *Google Docs*, a popular tool suite for collaborative editing of documents.[1]

Availability

We consider two aspects for the availability: (i) a tool should be easy to discover on the internet to reach a large audience, and (ii) its source code should be accessible for modification. We searched for URLs outside of the studies, *i.e.*, within download and usage instructions, or installation manuals. We contacted authors when a study failed to provide such URLs. In several cases, authors reported to us that their tools have become obsolete or were not maintained anymore. For instance, T80, which is a web browser extension, was reported to be obsolete. Similarly, T41, T44, and T68 are either not maintained or not published by authors for various reasons: the underlying technology being obsolete or due to non-disclosure agreements with their industry partners.

Websites. We were able to collect website URLs of 45 tools out of 112 (40%). The summary of information available on these 45 websites can be found in Table 22. The first column indicates the tool identifier. The

[1] *Google Docs*, accessed May 18, 2020, `https://www.google.com/docs/about/`

column "Tool introduction" indicates whether the website introduces the tool, the context of use, and any other related information. The column "Usage instructions/ feature introduction" indicates whether the website provides information regarding the features and usage of the tool. The column "Link to the source code repository" indicates whether the website contains a link to the source code repository such as GitHub. Finally, the column "Download link" indicates whether the website contains a link to download the tool. All "true" values are indicated with a checkmark (✓). A further 20 tools (18%) did not have any web page, and we failed to collect information for other 47 (42%) tools. We also found that T14, T18, T27, T31, T38, T40, T42, T51, T62, T68, T90, and T97 are web applications, but not hosted online by the authors. Instead, they need to be installed and run locally by the user. In conclusion, we found that a large number of RE tools lack any webpage, which can adversely affect their discoverability. It is evident from Table 22 that even though one-third of tools have dedicated webpages, still vital information, such as links to the source code repository, is often missing.

Openness. Figure 26 shows the evolution of RE tool openness over the years 2015-2019. It is evident that the number of open-source tools is

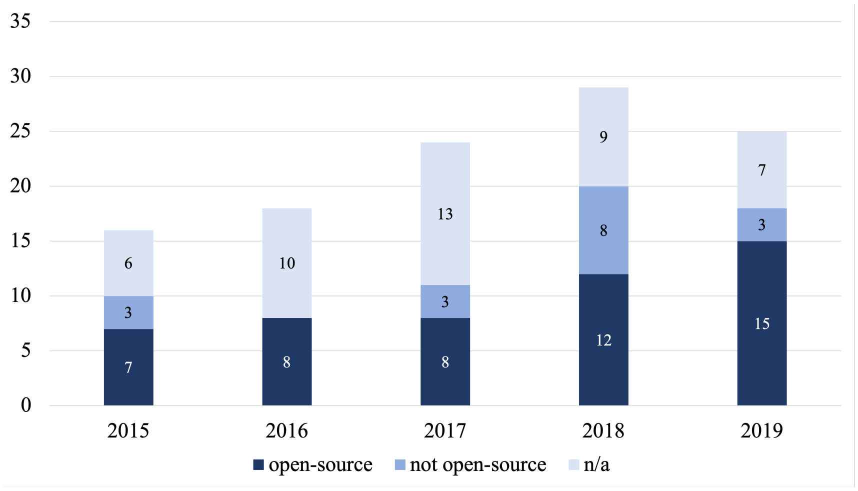


Figure 26: Evolution of open-source RE tools

steadily increasing. However, we found some special cases: T80, although open-source, does not have a source code repository, and T101 has only a private repository. For T92 we could not find any repository details at all. In summary, the authors of 50 tools (45%) reported their tool as being open-source, and 17 tools (15%) were reported as closed-source. It is noteworthy that the adaptation of open software development has

Table 22: Information available on tools' websites

Tool	Tool introduction	Usage instructions/ feature introduction	Link to the source code repository	Download link
T2	✓	✓	✓	✓
T6	✓	✓	-	-
T7	✓	✓	-	-
T11	✓	-	-	-
T12	✓	✓	✓	-
T13	✓	✓	-	-
T18	✓	✓	✓	✓
T19	✓	✓	✓	-
T22	✓	✓	-	✓
T23	-	-	-	-
T24	✓	✓	-	-
T25	✓	✓	-	-
T26	✓	✓	-	✓
T30	✓	✓	✓	✓
T33	✓	✓	-	✓
T34	✓	✓	✓	-
T38	✓	✓	-	✓
T39	✓	✓	✓	-
T43	✓	✓	-	✓
T44	✓	-	-	-
T46	✓	✓	-	✓
T53	-	-	-	-
T54	✓	✓	✓	✓
T55	✓	✓	✓	✓
T58	✓	✓	-	-
T60	✓	✓	✓	✓
T62	✓	✓	-	✓
T65	✓	✓	✓	✓
T66	-	-	✓	-
T67	✓	-	-	✓
T78	✓	-	-	✓
T79	✓	✓	-	✓
T85	✓	-	-	✓
T87	-	-	-	-
T93	✓	✓	✓	✓
T95	✓	✓	-	✓
T96	✓	-	-	-
T97	✓	✓	-	-
T98	-	-	-	-
T99	-	-	-	-
T100	✓	✓	-	✓
T101	-	-	-	-
T103	-	-	-	✓
T105	-	-	-	-
T112	✓	✓	-	-

increased in recent years.

Last activity on GitHub. In Figure 27 we show the evolution of Git activity over the years 2015-2019 for 42 Git repositories. The x-axis enumerates the publication years of the studies, whereas the y-axis shows the number source code repositories with last commit in a specific year. Out of 47 tools for which we have access to the source code, only 40 have Git repositories.

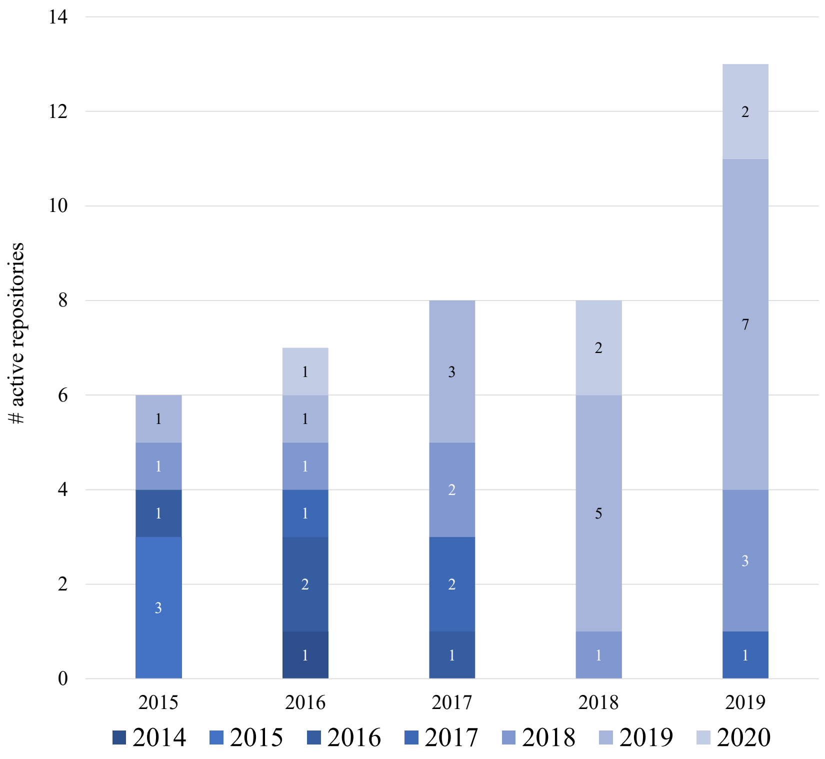


Figure 27: Evolution of project activity

Let us look at the third bar for the year 2017. Of the seven projects published in 2017 and had a source code repository accessible, only three had the last commit on their repository in 2019. The data was collected on February 4, 2021. We see that a very small number of tools are actively maintained over the years.

Number of contributors. Table 23 summarizes the number of code repositories against the number of contributors they have. As we can see, most of the tools are developed by a single developer. We further analyzed how well the 24 tools that are built by a single contributor are maintained. The corresponding results can be seen in Figure 28. The x-axis enumerates the publication years of the studies, whereas the y-axis shows the number of source code repositories with their last commit in a specific year. Let us look at the first bar for the year 2015. Of the three projects published in 2015 with a source code repository available, only one had the last commit in 2018.

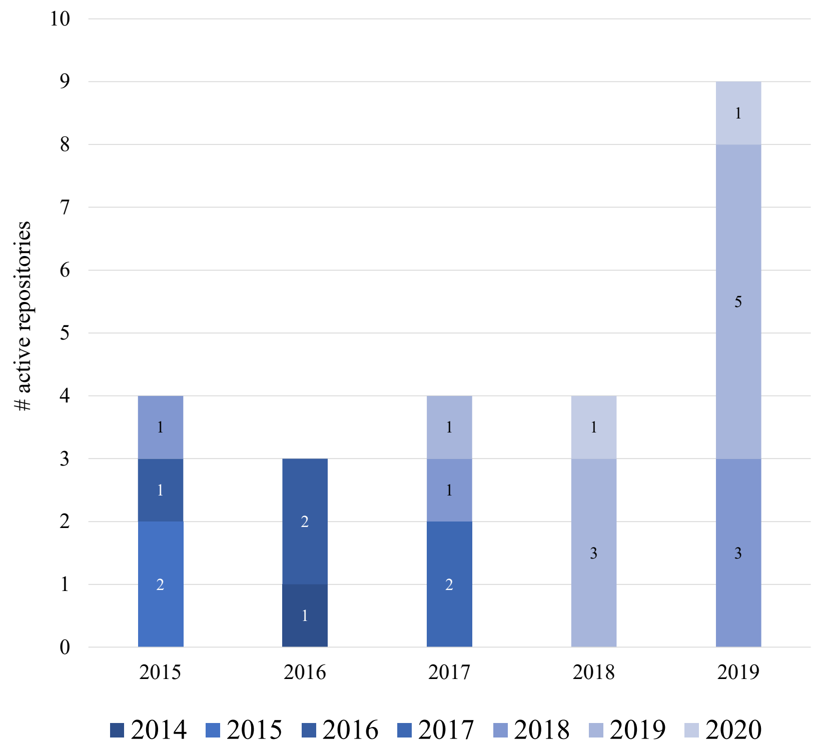


Figure 28: Project activity with single contributor

Our results indicate that most RE tools published by researchers are maintained only for a short duration. The reason, we speculate, is that they are built as throw-away prototypes (e.g., T4, T18).

Table 23: Relation between RE tool contributors and repositories

# Contributors	# Repositories
1	24
2	4
3	4
4	3
6	1
9	1
10	1
12	2

2.2 Software artifacts

The ability of software development environments or tools to automatically maintain the consistency of multiple and changing software-related artifacts is called round-trip engineering [115]. The existing research in round-trip engineering is limited to synchronizing modeling-related artifacts, *e.g.*, UML diagrams, with the source code [89, 141, 31, 27, 67]. However, numerous other artifacts are produced in other phases, *e.g.*, requirements gathering, design, and testing, of the software development lifecycle (SDLC) [60, 12, 40, 30]. These artifacts support specific activities, such as project planning or prototyping, and are created and managed in separate tools. For example, requirements-related artifacts, *e.g.*, user stories, are created and maintained in a project management platform like Jira [19, 69]. Modeling artifacts, such as UML diagrams, are created in tools like Lucidchart, whereas testing artifacts, such as behavior scenarios [125], are managed in a separate tool like Cucumber [4, 39]. When requirements change, multiple associated artifacts, the implementation, and eventually project documentation must be updated to ensure that all stakeholders access the current state of the project. However, in practice, an abundance of employed tools leads to inconsistencies among artifacts and the source code [8]. Project and requirements management tools, *e.g.*, IBM Rational DOORS and Enterprise Architect, manage a variety of artifacts, yet their support for round-trip engineering is limited to UML diagrams only [5, 1]. With the growing number of tools employed for artifact creation, management, and source code implementation, supporting round-trip engineering becomes difficult, eventually affecting project documentation [61, 60, 124].

Existing studies have discussed artifacts, especially those used in agile practices, both by analyzing the literature and surveying practitioners, thus giving us a comprehensive overview of the available artifacts [12, 124, 114, 113, 16]. We compiled a list of 62 artifacts mentioned in the existing literature and analyzed their flow within the software development lifecycle. The list of artifacts and methodology we followed to analyze those artifacts is sketched in subsection A.4.1 and subsection A.4.1, respectively. Next, we summarize the main results of our analysis.

2.2.1 Results

Figure 29 shows distribution of artifacts according to their formats.

It is evident from Figure 29 that most artifacts (39 out of 62) have a mixed format.

This finding suggests that if we wish to use an IDE to create and manage requirements using several artifacts, the IDE needs to provide capabili-

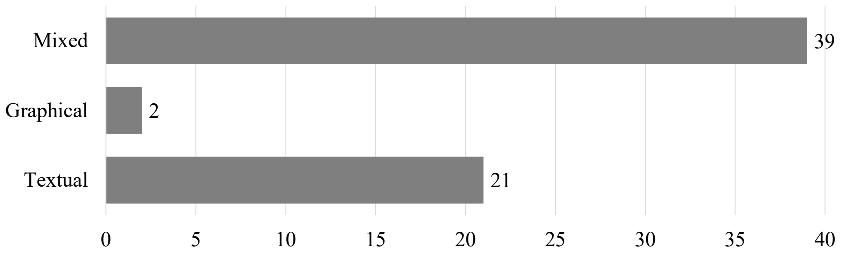


Figure 29: artifact distribution according to their formats

ties to render both textual and graphical characteristics. Specifically, it should have a graphical engine to create custom visualizations, and GUI components to create, render, and manipulate artifacts.

Figure 210 shows the flow of artifacts from the phases of origin to the target phases of usage. The left ("O_": the phase of origin) and the right side ("T_": the target phase) indicate the SDLC phases. The height of each node (bar) corresponds to the number of artifacts involved in a particular phase. Each artifact is connected with at least one node from the left and right ends. The connections are color-coded according to the distinct phases. The numbers next to the phase nodes signify the total number of artifacts involved in it.

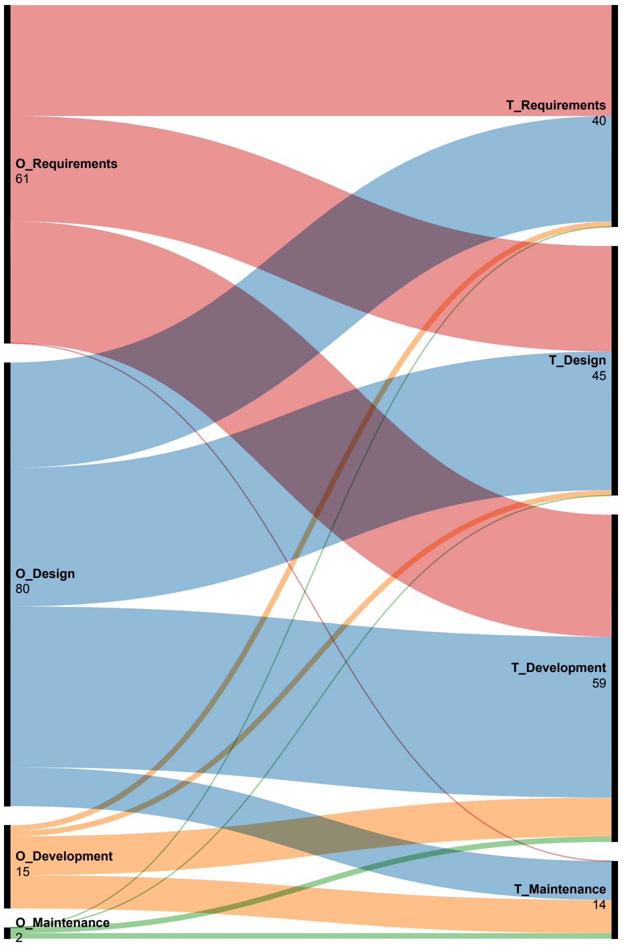
*Figure 210 shows that a significant number of artifacts (*i.e.*, 54 out of 62) are introduced in the early SDLC phases but, are in fact, consumed in the later SDLC phases, presumably by technical stakeholders.*

This finding suggests that artifacts have a longer lifespan. For example, a specific user story might be created early during development to record requirements and is used by developers later during implementation. Moreover, for several other reasons, such as archival purposes or retrospective analysis, it can be kept even after the implementation is finished. In successive development iterations, artifacts might undergo alterations and be used by different stakeholders for distinct purposes. If we wish to use an IDE to support agile development, we need to provide suitable GUIs for technical and non-technical stakeholders to create and manage artifacts. Additionally, the IDE must support the creation of suitable visual representations of artifacts for different stakeholders to accomplish their tasks.

2.3 BDD tools

BDD is an approach that enables domain experts to specify "live," executable, and testable requirements. Within BDD, domain experts specify

Figure 210: The flow of artifacts along SDLC phases



application behavior through scenarios that everybody in a team can understand [146]. They often leverage a constrained natural language, *i.e.*, Gherkin, to write behavior scenarios. For example, in Listing 1, we show a scenario written in Gherkin that asserts the sum of two numbers for an arithmetic calculator application to have a particular value.

```
Feature: Basic arithmetic operations
As a user
I want to use a calculator to add numbers
So that I do not need to add them myself
Scenario: Add two numbers 2 and 3
Given I have a Calculator
When I add 2 and 3
Then the result should be 5
```

Listing 1: A sample feature description with a scenario

A typical Gherkin template splits a scenario into three core steps: *Given* (*i.e.*, a context assumed for this scenario execution), *When* (*i.e.*, an action or event that occurs in the given context), and *Then* (the expected outcome of the system for the provided action and context). A step can have additional context, expressed in the template by the keyword *And*. Apart from these four keywords, Gherkin contains several other keywords, such as *Background* or *Rule*. The BDD frameworks automatically tie the steps in scenarios to acceptance test cases (also called step definitions, glue code, or fixtures) to verify the specified functionality. Listing 2 shows the corresponding glue code for the scenario in Listing 1. The developers need to fill in the body of glue code methods.

```
public class CalculatorRunSteps {
private int total;
private Calculator calculator;
@Before
private void init() {
total = 999;
}
@Given("I have a calculator")
public void initializeCalculator() throws Throwable {
calculator = new Calculator();
}
@When("I add {int} and {int}")
public void testAdd(int num1, int num2) throws Throwable {
total = calculator.add(num1, num2);
}
@Then("the result should be {int}")
public void validateResult(int result) throws Throwable {
Assert.assertThat(total, Matchers.equalTo(result));
}
}
```

Listing 2: Glue code for the scenario from Listing 1

Next, developers implement the logic for the calculator application:

```
public class Calculator {
public int add(int a, int b) {
return a + b;
}
}
```

Listing 3: Implementation for the functionality from Listing 1

Finally, when domain experts execute the acceptance tests, the BDD frameworks present them with the test run status, *i.e.*, success or failure.

In a recent survey, software engineers and business analysts highlighted several shortcomings of current BDD practices [21]. They complained that they must write numerous scenarios with minor variations in input parameter values. Additionally, they must also specify the test assertions. They mentioned that when requirements change, a lot of manual effort is needed to maintain the textual scenarios and to manually propagate the changes to acceptance tests, leading them to perceive BDD as only an additional task to writing unit tests [21, 149]. To reduce the redundancy in Gherkin files and improve their readability, keywords, such as *Scenario Outline*, and features, such as data tables, were introduced in the year 2009 [2]. In Gherkin, a *Scenario outline* is parameterized using *Examples* data tables. In Listing 4, we see how a data table can be used to test several combinations of input numbers against the corresponding result.

```
Scenario Outline: Sample arithmetic additions
  Given I have a Calculator
  When I add "<num1>" and "<num2>"
  Then the result should be "<result>"

  Examples: Numbers
    | num1    | num2  | result    |
    | 1       | 3     | 4         |
    | 5       | 8     | 13        |
    | 7       | 2     | 9         |
```

Listing 4: A sample data table

Similarly, data tables can be passed into a step as an input data structure to improve the readability. Nevertheless, practitioners still have difficulty maintaining Gherkin files a decade after these features were introduced.

To explore the specifics of the current BDD practice, we conducted two studies:

- *An exploratory study*: we analyzed the contents of 1,572 Gherkin files extracted from 23 open-source projects. In section A.7, we describe the methodology we followed.

- *A tool analysis*: we analyzed 13 BDD tools that are open-source and actively maintained on GitHub to provide documented proof of our claims of the currently adopted BDD workflow shortcomings. In particular, we studied how the IDE integration enables behavior specification and verification for non-technical stakeholders. We analysed Cucumber [39], JBehave [136], Concordion [134], SpecFlow [138], Spock [137], RSpec [132], MSpec [87], LightBDD [80], ScalaTest [140], Specs2 [122], JGiven [131], phpspec [105], and Gauge [135]. In section A.8, we describe the methodology we followed.

Next, we summarize the results of both studies.

2.3.1 Results for the exploratory study

Meta statistics

Table 24 summarizes the meta information for the selected repositories. We used the following abbreviations: *m1*: number of Gherkin files, *m2*: date of the last commit on the repository (*a date*), *m3*: date of the last commit on the Gherkin file (*a date*), *m4*: number of contributors to the repository, *m5*: number of contributors to the Gherkin files. Columns *m1-5* present results for the corresponding metrics. It is evident from Table 24 that 11 out of 23 (about 48%) repositories have fewer than 20 Gherkin files, whereas only three repositories have more than 100. Notably, one of those repositories is the Cucumber project itself, making the high number not so surprising. The number of contributors to the identified repositories is also quite diverse, ranging from 3 to 736. However, the number of contributors to the Gherkin files is quite low, ranging from 6 to 15, meaning only a small fraction of all contributors are involved in modifying them. Most of the repositories (about 83%) have rather recent commits (in the year 2021), meaning these are actively maintained. However, only about 39% of the total repositories had last commits on Gherkin files in 2021. Importantly, a similar number of repositories have the last commit on Gherkin files before 2019, which makes us conclude:

Although repositories are maintained actively, teams might stop using BDD in their project for some reason.

Table 24: The meta-level details of the selected repositories

Repo name	Index	*m1*	*m2*	*m3*	*m4*	*m5*
eugenp/tutorials	R1	15	08 Jun 2021	30 May 2021	736	06
neo4j/neo4j	R2	26	07 Jun 2021	04 Mar 2021	207	15
geoserver/geoserver	R3	04	08 Jun 2021	19 Sep 2017	266	01
apache/servicecomb-pack	R4	30	03 Apr 2021	16 Mar 2021	56	04
microservices-patterns/ftgo-application	R5	02	02 Jun 2021	10 Jul 2018	3	01
apache/tinkerpop	R6	59	07 Jun 2021	18 Jun 2021	142	06
iotaledger/iri	R7	05	18 Aug 2020	07 May 2020	58	04
SmartBear/soapui	R8	31	09 Dec 2020	07 Jul 2014	63	08
w3c/epubcheck	R9	36	15 Mar 2021	26 Feb 2021	11	03
aws/aws-sdk-java-v2	R10	53	07 Jun 2021	14 Aug 2018	70	01
bugsnag/bugsnag-android	R11	48	07 Jun 2021	15 Jun 2021	128	06
blox/blox	R12	10	12 Mar 2018	12 Feb 2018	22	03
ddd-by-examples/factory	R13	02	24 Apr 2021	22 Dec 2017	06	01
FluentLenium/FluentLenium	R14	11	08 Jun 2021	14 Jul 2019	62	02
AppiumTestDistribution/AppiumTestDistribution	R15	03	08 May 2021	19 Dec 2020	35	04
mzheravin/exchange-core	R16	02	25 Apr 2021	07 Jun 2020	05	02
iriusrisk/bdd-security	R17	11	08 Aug 2018	24 May 2018	10	01
jbangdev/jbang	R18	18	07 Jun 2021	24 May 2021	56	04
SoftInstigate/restheart	R19	31	07 Jun 2021	11 Jun 2021	27	04
intuit/karate	R20	394	24 May 2021	16 Mar 2021	54	09
cucumber/common	R21	439	07 Jun 2021	-	111	10
cucumber/cucumber-jvm	R22	92	06 Jun 2021	-	225	06
JetBrains/intellij-plugins	R23	103	07 Jun 2021	09 Apr 2021	208	06

In Table 25, we report values for various parameters derived for metrics *m6-8*. We used the following abbreviations: *a*: a total number of Gherkin files containing tables, *b*: an average number of LoC in all Gherkin files, *c*: an average number of LoC in all Gherkin files with tables, *d*: an average number of LoC in scenarios in all Gherkin files, *e*: an average number of LoC in scenarios in all Gherkin files with tables *f*: an average number of tables per Gherkin file, *g*: an average number of scenarios per Gherkin file, *h*: an average number of scenarios per Gherkin file with tables. It is

Table 25: General statistics across all Gherkin files

a	b	c	d	e	f	g	h
590	38.53	73.41	35.53	71.11	11.41	3.47	5.85

evident from Table 25 that each Gherkin file, on average, contains about 3.47 scenarios. The average use of the *Background* keyword (0.25 times per Gherkin file) seems reasonable, meaning a negligible number of steps were common among a very few scenarios present in any Gherkin file. The average use of the *Feature* keyword (0.94 times per Gherkin file) also seems reasonable as each Gherkin file will typically test a single feature. A total of 590 Gherkin files (*i.e.*, about 37.5%) contain tables. On average, each file contained 11 tables. Notably, each table contains about 2.6 rows and 1.7 columns, on average, which leads us to conclude:

Data tables are not widely used yet.

Table 26 further summarizes results for individual repositories. We used the following abbreviations: *a*: an average number of LoC per Gherkin file, *b*: an average number of LoC in scenarios per Gherkin file, *c*: an average number of LoC per Gherkin file with tables, *d*: an average number of LoC in scenarios per Gherkin file with tables, *e*: an average number of tables per Gherkin file, *f*: an average number of scenarios per Gherkin file, *g*: an average count of the *Scenario outline* keyword per Gherkin file, *h*: an average count of scenarios per Gherkin file with tables, *i*: an average count of the *Scenario outline* keyword per Gherkin file with tables, *j*: an average count of the *Given* keyword per Gherkin file, *k*: an average count of the *When* keyword per Gherkin file, *l*: an average count of the *Then* keyword per Gherkin file, *m*: an average count of the *And* keyword per Gherkin file, *n*: an average count of the *Feature* keyword per Gherkin file, *o*: an average count of the @ (*i.e.*, tags) keyword per Gherkin file, *p*: an average count of the *Background* keyword per Gherkin file, *q*: an average count of the *Examples* keyword per Gherkin file. It is evident from Table 26 that the average number of tables per specification is higher than expected because only four repositories, *i.e.*, R16, R21, R13, and R2,

Table 26: Repository-wise details

Index	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
R1	13.93	11.6	19.6	15.8	2	2.33	0.07	2.6	0.2	1.8	2.4	2.4	0.8	1	0.2	0.13	0.07
R2	217.15	215.11	217.15	215.11	13.04	12	0.22	12	0.22	7.67	12.89	12.3	22.59	1	0.33	0.33	0.22
R3	62	58.75	-	-	0	4.25	0	-	-	4.25	4.25	4.25	16.25	1	0.65	0	0
R4	25.97	24.97	25.97	24.97	3.63	1.17	0	1.17	0	1.63	1.17	2.57	4.47	1	0	0	0
R5	15	12	-	-	0	1.5	0	-	-	4.5	1.5	2.5	2	1	0	0	0
R6	157.39	156.39	160.5	159.5	10.81	11.95	0	12.43	0	11.95	11.64	11.93	19.47	1	0	0	0
R7	75.4	71.6	118.5	116.5	5.5	4.8	0	7.5	0	4.8	3.4	6.6	6.4	1	0	0	0
R8	31.87	29.45	29	26.67	2	4.39	0.45	1	1.67	3.1	2.97	3.16	12.03	0.94	1	0.03	0.45
R9	97.69	89.58	145.38	137.54	1.54	21.22	0	32	0	1.89	21.22	21.75	23.69	1	0	1	0
R10	10.51	8.42	10.51	8.42	1.17	2	0	2	0	0	2	2	0.23	1	1	0	0
R11	38.2	37.04	50.17	49	2.17	2.49	0	4	0	0	2.55	2.29	28.94	1	0.33	0	0
R12	23.4	19.1	29.8	24.8	4.4	2.9	0	3	0	2.4	3	3	3.2	1	2.2	0.3	0
R13	84.5	58.5	84.5	58.5	16.5	4.5	0	4.5	0	7	6	16.5	0.5	1	0	0	0
R14	8.82	7.82	-	-	0	1.91	0	-	-	1.55	1.91	1.91	0.36	1	0.18	0	0
R15	12.67	7.67	17	12	1	1.33	0.33	1	1	1	1.67	1.33	0.67	1	1	0	0.33
R16	89	82.5	89	82.5	24	1.5	0.5	1.5	0.5	3.5	10.5	10.5	18.5	1	2	0.5	0.5
R17	42.09	37.18	23.4	18.8	1.4	3.82	0.64	1.2	1	2.09	3.64	4.73	13.36	1	5.27	0.27	0.64
R18	14.83	13.22	-	-	0	3.17	0	-	-	0	2.94	3	0.28	1	0.06	0.28	0
R19	101.87	90.16	-	-	0	6.03	0	-	-	12.1	11.61	11.42	26.13	1	1.32	1	0
R20	19.02	16.04	45.47	41.34	2.07	1.92	0.19	2.87	1.18	0.81	0.88	1	1.74	1	0.44	0.44	0.22
R21	39.78	36.6	80.55	78.72	21.04	3.11	0.43	5.92	0.94	3.54	0.06	0.05	8.71	0.85	0.66	0.19	0.48
R22	11.48	9.06	21.38	18.88	2.31	1.49	0.27	2.16	0.78	1.62	1.16	1.28	0.3	0.95	0.35	0.13	0.39
R23	8.51	7.07	15.17	14.06	1.2	1.39	0.29	2.43	0.74	2.1	0.54	0.22	0.44	0.96	0.15	0.04	0.3

contain significantly more number of tables per Gherkin file than other repositories, on average. The average number of LoC in Gherkin files varies greatly among selected repositories: from 8.51 to 217. The average number of scenarios per Gherkin file also varies significantly across those containing tables. For example, if we compare repository R9 that has on average 21.2 scenarios per Gherkin file without tables, in fact, has 32 scenarios per Gherkin file containing tables. Finally, although the average number of scenarios per Gherkin file is low, *i.e.*, 3.47 times (for the top three repositories, *i.e.*, R9, R19, R8, with a maximum number of scenarios per Gherkin file, which is 21.2, 6.03, and 4.39 times), we expected more use of the @ keyword (*i.e.*, 0, 1.32, and 1 times), meaning that users did not particularly attempt to optimize the test run time by marking a subset of tests to be run independently. Not many practitioners have used keywords, such as *Examples* (used on average 0.25 times per Gherkin file), meaning there is not much test automation used. The use of the *Scenario outline* keyword is negligible among all the repositories, *i.e.*, less than 0, on average.

2.3.2 Results of the BDD tool analysis

We analyzed features of 13 BDD tools that support the end users in specifying and verifying application behavior. Next, we summarize our findings of our analysis.

Type of input

Specifications are written either textually (5 tools) or as test cases enhanced with annotations, such as `[Given]` (8 tools). The textual specifications are written either with Gherkin syntax, in a Markdown format, or a combination of both. Data tables with input and expected output values for behavior tests are supported in a total of 4 tools, which means

Data tables are not supported universally across all tools.

Support for parameterized scenarios

We observed that only primitive types, such as strings or numbers, are allowed as inputs to the scenarios. Data tables help to specify input parameters concisely. However, how helpful the data tables are to specify complex domain objects with numerous attributes has not yet been studied.

Specification interface

All the analyzed tools support textual specification only;

None of the tools allows specifications to be composed in any other way, e.g.*, graphically.*

Type of output

All analyzed tools provide two ways to output their results: (i) test results that indicate how many tests are passing and failing, and (ii) test reports that can be customized with formats (*i.e.*, charts and graphs) and color schemes (*i.e.*, indicating passed and failed tests in different colors), meaning

There is no other alternative for non-technical stakeholders to verify whether developers implemented the correct behavior in the source code.

2.4 Related work

2.4.1 RE tools

To the best of our knowledge, there is no SLR that reviews RE tools or those specific to a single RE activity. However, there are several SLRs published from the RE community that cover various aspects and branches of RE:

The work of Davis *et al.* reports the effectiveness of various RE techniques [42]. The work of Pacheo *et al.* surveys studies on stakeholder identification methods in requirements elicitation [95]. Achimugu *et al.* surveyed studies on requirements prioritization techniques [7]. Schoen *et al.* surveyed publications in agile requirements engineering focused on stakeholder and user involvement [114]. None of the above studies discuss or compare available tool support for any of the RE techniques or activities they considered.

The work of Alves *et al.* focuses on software product line engineering and reports open problems in it from the RE point of view [11]. Of the studies they considered, 43% proposed no tool support. Similarly, Inayat *et al.* surveyed studies that discuss agile RE with the motivation of mapping evidence about how available RE practices are adopted by agile teams [66]. Of the the reviewed studies, 18% were empirically evaluated tools. Recent work of Khan *et al.* surveyed publications from crowd-based requirements engineering, which included seven RE tool demos [72]. However, none of them performed an assessment or evaluation of the found tools. Instead, they express it as a need for future research.

The closest work in literature to ours is that of Carrillo de Gea *et al.* [44, 45] and Shah *et al.* [117]. Shah *et al.* in their mapping study present a comparison of 31 RE tools. They compare the tools based on the RE activity they support, the geographic origin of the study, and the number of RE activities they cover. They searched the literature in four digital libraries with a search string ("Software requirement Engineering tools" or "Software Elicitation tools"). They claim to evaluate tools according to their performance, however, there is no discussion about it in their work to understand the specifics of the term "performance." Carrillo de Gea *et al.* present an overview of 38 RE tools, specifically proprietary tools. Their work is not an SLR, instead, they performed an online survey with 146 questions based on ISO/IEC TR 24766:2009 in which vendors such as IBM and IRqA participated.[2] The objective of their work was to evaluate RE tools and technologies with different use cases to report if they are adequate to cater to the present needs of software development practices. They discovered that most of the tools they evaluated work well for elicitation, and there is still scope to improve the modeling tools.

2.4.2 Software artifacts

Only a handful of studies, to our knowledge, have discussed the characteristics of the artifacts, concerning their fitness for distinct stakeholders or fulfilling specific purposes [103, 17, 48, 49]. A few studies have discussed classification schemes for artifacts. These include: (1) considering socio-technical aspects (*e.g.*, target audience) of artifacts [103, 48, 49], and (2) considering the abstraction levels based on phases of SDLC [17, 148]. Several other studies have proposed solutions to establish traceability among numerous artifacts [61, 41, 107, 15, 10, 129, 118]. The existing classification schemes do not characterize artifacts according to their physical properties. To build usable user interfaces for artifact creation and manipulation, we need to understand the format, nature of artifacts, and their relation to one another. In particular, we consider characteristics that would help us to: (1) understand artifact flow within SDLC, and (2) decide whether and how can we create these artifacts in an IDE to be usable by distinct stakeholders.

2.4.3 BDD tools

BDD exploratory studies

Several studies in recent years have proposed approaches and techniques to automate the BDD process. Soeken *et al.* proposed a technique to semi-automatically generate step definitions and code skeletons from scenarios given in natural language [119]. Patkar *et al.* analyzed the features of the current BDD tools and proposed to specify application behavior through in-IDE graphical interfaces [97]. With their approach, they engage non-technical stakeholders in the BDD process equally. Binamungu *et*

[2]*ISO/IEC TR*, accessed 20 April, 2020, `https://www.iso.org/standard/51041.html`

al. presented a dynamic tracing based approach for detecting duplication in BDD suites [20].

Apart from these studies, several empirical studies shed light on various aspects of the BDD. Binamungu *et al.* surveyed 75 BDD practitioners to understand the extent of BDD use, its benefits and challenges, and specifically the challenges of maintaining BDD specifications in practice [21]. Their results showed that BDD specifications suffer from maintenance challenges due to the huge size of the BDD suites. They also conducted another survey with BDD practitioners, to hear their opinions on the quality criteria for the BDD specifications established by the authors themselves [22].

Yang *et al.* from 59,933 open-source Java projects retrieved 133 projects containing at least one *.feature* file [147]. They figure out whether and how accurately could they identify co-changes between *.feature* and source code files when either of those changes. In this study, they used natural language processing to check both *.feature* files and the source code files to detect the occurrences of common keywords. They did not study the step definitions, and their results are specific to Cucumber related projects.

Zampetti *et al.* analyzed 20 Ruby projects shortlisted from the top 50,000 projects— ranked in terms of several stars— hosted on GitHub for the five most popular programming languages, *i.e.*, Java, Javascript, PHP, Python, and Ruby [149]. Their goal was to study the extent to which open-source projects use BDD-related frameworks, *i.e.*, the percentage of projects that use one of the several BDD frameworks. They also surveyed 31 developers to understand how these developers use BDD frameworks in practice. They observed a co-evolution between scenarios and fixtures, and source code in about 37% of the projects. Specifically, the authors discovered that changes to scenarios and fixtures often happen together or after changes to source code. Moreover, survey respondents indicated that, while they understand the intended purpose of BDD frameworks, most of them write tests while/after coding rather than strictly applying BDD.

Neither of these studies shed light on the specifics of the Gherkin files. The data published by Yang *et al.* contains repositories with fewer than ten stars and consists of mostly small personal projects. Zampetti *et al.* offer to download their dataset. Regrettably, it consists of only processed data and not the data from the fetched repositories.

BDD tools analysis

Only a few prior studies have evaluated BDD tools. Previously analyzed tools such as StoryQ [133], JDave [68], NBehave [91], Easyb [130], and BDDfy [139] are either obsolete or no longer maintained [94, 78]. Lenka *et al.* analyzed five BDD tools and classified them either as testing tools or test automation frameworks, essentially supporting the view of BDD tools as being testing tools [78]. Solis *et al.* analysed seven BDD tools according to six parameters, such as the supported programming languages and supported software development phases [120]. They observed poor support for BDD in the planning phase, *i.e.*, the analysed tools did not support the creation of features or user stories. Okolnychyi *et al.* analyzed five BDD tools to characterize their support for BDD in terms of ubiquitous language creation and automated scenario execution [94]. They compared the tools based on their primary target users and specific tool features, such as support for mocking third-party libraries. They observed that the support for ubiquitous language definition is limited. These studies do not establish criteria to measure to what degree BDD tools enable collaboration among both technical and non-technical stakeholders.

2.5 Summary and conclusion

Our analysis of state of the art RE and BDD tools, current BDD practices from open-source projects, and software artifacts help us explain why there is a lack of agile assistance for requirements specification, verification, and management. Our main findings suggest that numerous software-related artifacts are created in several distinct tools. There is an increasing trend of developing specialized tools to support RE activities, which can lead to traceability issues among produced artifacts. Our analysis of BDD tools shows that non-technical stakeholders are required to use an IDE to verify the application behavior but they do not have an adequate support as an IDE is not adapted for non-technical stakeholders. These results give us confidence in imagining an agile development platform to support various stakeholders with software engineering activities, specifically, specification, validation, and management of requirements, concurrently with the source code.

Chapter 3

Citizen requirements

In this chapter, we revisit three approaches from the existing research, discuss their underlying ideas, and show how they can be adapted while re-imagining a collaborative agile development platform. We discuss the building blocks, which can be implemented in any IDE.

3.1 Background

3.1.1 The naked objects pattern

"Naked objects" is an architectural pattern wherein core business objects, such as Customer, Product, and Order, are exposed directly to the user through the auto-generated user interfaces [101]. User interfaces are generated at run-time based on the capabilities of the core business objects. In software designed using the naked objects pattern, all user actions involve explicitly invoking methods on business entity objects. The naked objects pattern contributes to the agility of the development process in two ways [102]:

- It facilitates communication between the developers and users during requirements analysis as the business objects provide a common language. The same idea is central in other approaches, such as domain-driven design (DDD), where Evans promotes using a ubiquitous language that appears in all aspects of the project: communication, test cases, and the source code [54].
- It speeds up the development cycle as developers are not required to design and implement the user interface. They only have to design and implement the business object classes and their encapsulated business methods.

The naked object framework has been designed specifically to support the naked objects pattern [100]. Most IDEs support the creation and manipulation of business objects programmatically. Very few IDEs, such as BlueJ,[1] enable non-technical stakeholders to create and manipulate objects interactively. For example, in BlueJ, objects can be dynamically created, the contents of fields are displayed, and their methods can be invoked through provided GUIs. Like in BlueJ, the auto-generated GUIs in the naked objects framework are generic and cannot be customized to adapt to a particular application domain. Nevertheless, from the naked objects pattern, we adopt the idea that requirements can be discussed at the level of business objects, and end users use GUIs to manipulate core business objects directly.

3.1.2 Low-code development platforms

Although the term "low-code development platform" is very recent, it has its roots in earlier ideas, such as the end user programming, visual programming, and rapid application development (RAD) tools [110]. These ideas proposed activities and tools to involve people with less or no programming experience in the software development process. For example, RAD tools combine fourth-generation languages, GUI builders, database management systems, and computer-aided software engineering tools, and allow changes to prototypes to be made in-situ at user-developer meetings [18]. Similarly, end user programming environments are heavily graphical in nature.

With LCDPs, end users without programming skills create software using GUIs. LCDPs can lower the initial cost of setup, training, deployment, and maintenance. They have proven to reduce the amount of traditional hand-coding, enabling accelerated delivery of business applications [111]. LCDPs leverage model-driven engineering principles and cloud infrastructures, automatic code generation, declarative and high level, and graphical abstractions to develop entirely functioning applications [112]. From LCDPs, we adopt the idea that end users can be involved in the development workflow, in fact, can be supported to build certain types of applications themselves, provided the right GUIs exist.

3.1.3 Moldable development

With the moldable development approach, an IDE is modeled as a set of interconnected context-aware tools [35]. Such tools (*e.g.*, object inspector, debugger) are aware of the current development context, enabling rapid customization to new development contexts. Authors of moldable development argue that IDEs often focus on low-level programming tasks that promote code rather than data. Most IDEs suppress customization, offering limited support for informed decision-making during software development. In contrast, the moldable development approach helps to im-

[1]`https://www.bluej.org/`

prove program comprehension as context-aware tools can directly answer domain-specific questions. From the moldable development approach, we adopt the idea of customizing an IDE to support a range of software engineering tasks. We demonstrate that the moldable development approach can be leveraged to enable distinct stakeholders in specifying requirements at different levels of detail and verify those with representations specific to a particular application domain.

3.1.4 First-class entities

In programming language design, a first-class entity is an entity (*e.g.*, run-time object, value, type) that supports all operations (*e.g.*, being passed as an argument, returned from a function, assigned to a variable) generally available to other entities. Kiandl and Glinz independently envisioned representing and organizing project requirements as run-time objects [58, 70]. With their approach, one classifies and organizes requirements using classes and enjoys various benefits of object-oriented design. For example, using inheritance, one decomposes requirements into a hierarchical object structure. Such requirements hierarchies feature abstraction and information hiding, and exhibit the benefits of comprehensibility for various stakeholders: non-technical stakeholders benefit from high-level requirements, whereas technical stakeholders benefit from low-level ones, such as use cases. In this thesis, we propose that not only requirements, but also other software-related artifacts should be created as first-class entities directly in an IDE.

3.2 Building blocks

In a previous work, we revisited several ideas and techniques for requirements elicitation, *e.g.*, SWOT analysis and Personas, and adapted them in an assisted workflow for a web application [116]. The idea was to enable various stakeholders in specifying requirements– with underlying contexts and goals– whenever they wish in a single platform. We argued that while eliciting requirements for niche application domains, it is possible to get insights into domain-specific matters literally anytime and not necessarily only during the requirements gathering workshops or interviews. Consequently, the web platform guides non-technical stakeholders through the elicitation process without having to work directly with requirements engineers. Requirements engineers can go through the information entered by other stakeholders anytime. This evolving information allows requirements engineers to extract tacit knowledge and overall, serve as a vision of the application to be built.

In this doctoral thesis, we propose a similar approach, which we call citizen requirements, as to maintain requirements in a single platform, allow them to evolve, and support other phases of RE. But, instead of using

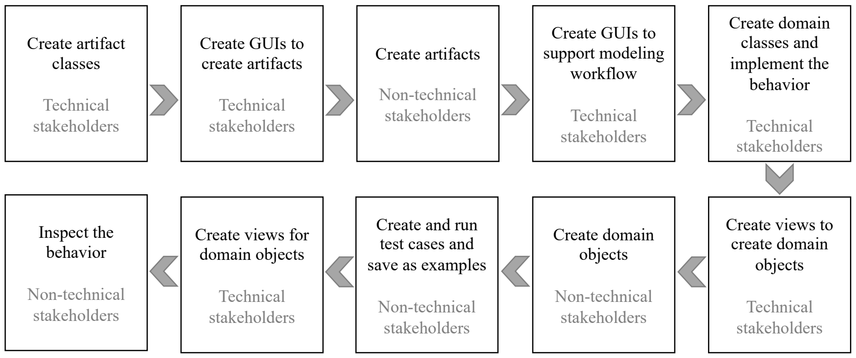

Create artifact classes
Technical stakeholders
Create GUIs to create artifacts
Technical stakeholders
Create artifacts
Non-technical stakeholders
Create GUIs to support modeling workflow
Technical stakeholders
Create domain classes and implement the behavior
Technical stakeholders
Create views to create domain objects
Technical stakeholders
Create domain objects
Non-technical stakeholders
Create and run test cases and save as examples
Non-technical stakeholders
Create views for domain objects
Technical stakeholders
Inspect the behavior
Non-technical stakeholders

Figure 31: Citizen requirements potential workflow

a web application, we propose to use an IDE. The main advantage of using an IDE is that the first-class citizen requirements are always up-to-date and verifiable, evolving concurrently with the implementation. As various stakeholders are provided with the right interfaces to work within a single platform, we can streamline an agile development workflow with faster implementation and feedback loops. There are two main parts of our approach: creating first-class artifacts in an IDE and verifying requirements collaboratively. To bring such an idea into existence, we need the following three building blocks: *Examples*, *GUIs*, and *Views*. Please note that some of these building blocks are research ideas already discussed in the existing literature. We have reused them to build a proof of concept to answer our research question. Specifically, we used the *GUIs* in combination with *views* to specify requirements— we use them to create and maintain software artifacts and business entities, natively in IDE. For verifying the behavior, we used all three building blocks. In Figure 31, we show a potential workflow within an IDE. Following this workflow, one would end up with validated requirements maintained directly in an IDE, which are ready to be pushed with the source code to source code management tools, such as GitHub.

3.2.1 Examples

Examples are nothing but test cases, which return intermediate business objects. Gaelli *et al.* first leveraged code examples in proposing an approach to unit testing, wherein they compose unit tests of code examples [55, 63, 76]. Their approach of example-driven testing states that fixture instances are valuable objects, and hence, to be reused and treated first-order by a testing framework. In the Glamorous toolkit, the term “example” is generally used to refer either to the example method or the example object produced. *Examples* are annotated with a `gtExample` pragma. The *example* in Listing 5 creates and returns an `Order` with a

`Coffee`, checks whether the `Order` object contains a `Coffee` instances, and returns the resulting `Order` object for further inspection.

```
    orderWithCoffee
    <gtExample>
    <label: 'create order with coffee'>
    <description: 'Create an order'>
    |order coffee|
    coffee := Coffee new.
    order := self emptyOrder add: coffee.
    self assert: order size equals: 1.
    ˆ order
```

Listing 5: An example method that creates an order with coffee

Unlike tests, *examples* can be arbitrarily chained together so that domain objects flow through a series of examples emulating complex scenarios. For instance, Listing 5 creates an `Order` with a `Coffee` by first calling Listing 6 that creates an empty `Order` object. This makes *examples* reusable.

```
1    emptyOrder
2    <gtExample>
3    <label: 'create an empty order'>
4    <description: 'Create an empty order'>
5    | order |
6    order := Order new.
7    self assert: order size equals: 0.
8    ↑ order
```

Listing 6: An example that creates an empty order

One implication of such chaining is that we can order *examples*, and when multiple *examples* fail, we can point to the most specific failure. *Examples* are useful to create both simple business objects, *i.e.*, objects that do not contain or require other business objects (*e.g.*, an empty `Order` object), and complex business objects, *i.e.*, run-time objects that contain other domain objects (*e.g.*, an `Order` with a `Coffee` object).

Software testing is the act of examining the artifacts and the behavior of the software under test by validation and verification. Often, writing test cases is assumed to be a task of testers or developers. If we want to involve domain experts in validating requirements, assuming that they might lack any technical knowledge, we need to provide them with the ways for writing, or better composing, test cases or *examples*. Within citizen requirements, we use *examples* in combination with GUIs to support non-technical stakeholders in verifying the application behavior. We use GUIs to create both simple and complex business objects, and to compose them in *examples*. To make generated *examples* understandable for non-technical stakeholders, a description can be given that is understandable by all the stakeholders. A description is provided by using a dedicated pragma, see line 4 of Listing 6. In chapter 5, we describe this process in detail.

3.2.2 GUIs

In general, GUIs enable end users to interact with the low-level business objects. For instance, imagine a web application. Low-level business objects are exposed to the the end user on the frontend through GUIs. These GUIs enable end users to perform standard manipulation operations, such as create, read, update, and delete (CRUD), on the business objects. However, building a fully functioning application and GUIs for object manipulation is quite expensive, especially when requirements change frequently. We believe that, if we provide GUIs for object manipulation directly in an IDE, we could enable fast feedback loops with non-technical stakeholders.

Unlike the naked objects framework, the Glamorous toolkit is an IDE, which is a fully reflective environment. It is built using a graphical framework (*i.e.*, Bloc) that enables the creation of customizable GUIs for various types of objects. Within citizen requirements, we use GUIs to support non-technical stakeholders in creating artifacts and business objects as first-class entities. It is primarily developers' responsibility to create tailored in-IDE GUIs. This approach can only succeed if in-IDE GUIs are created at a cheaper price in terms of efforts. In this thesis, we only show a proof of concept of using in-IDE GUIs to support agile development and demonstrate how a custom development workflow engages non-technical stakeholders in an IDE. In future, we need to provide easier ways (*e.g.*, an API) to support non-technical stakeholders specifying various kinds of GUIs, which will bring our idea even closer to that of LCDPs.

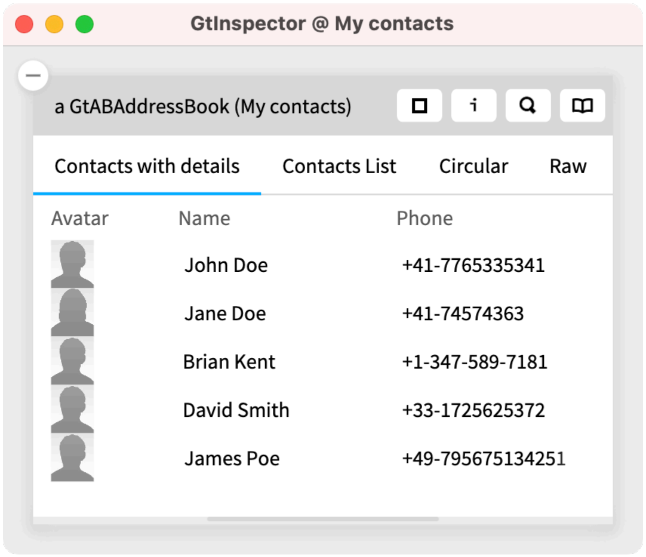
3.2.3 Views

Most modern IDEs have an object inspector that allows developers to inspect run-time objects. They show a standard representation for all types of objects regardless of the underlying domain, which usually includes various attributes of a particular object and the corresponding values [36]. This representation is primarily targeted towards developers. Non-technical stakeholders, on the other hand, often deal with business objects using a domain-specific representation. For example, in a web application, GUIs are used to display low-level business objects with a representation that is understandable for the end users. Imagine showing a clickable list of products with only essential details to the end users instead of showing a raw JSON representation of the product list. We can safely assume that most end users will only be able to work with a product list and not the raw JSON.

A *view* in the Glamorous toolkit is a domain-specific representation of an object. It is essentially a piece of code that generates a domain-specific visual representation for an object. The piece of source code in Listing 7 generates a domain-specific representation for a contact list as shown in Figure 32.

```
gtViewContactsOn: aView
<gtView>
↑ aView columnedList
    title: 'Contacts with details' translated;
    priority: 5;
    items: [ self contacts ];
    column: 'Avatar'
        icon: [ :aContact | aContact avatar asElement
        asScalableElement size: 32 @ 32 ]
        width: 75;
    column: 'Name' text: [ :aContact | aContact fullName ];
    column: 'Phone' text: [ :aContact | aContact telephone ]
```

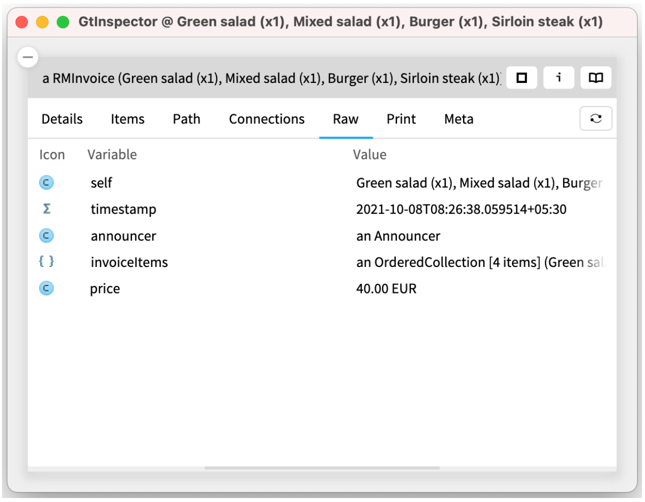
Listing 7: A sample source code for a view



Avatar	Name	Phone
	John Doe	+41-7765335341
	Jane Doe	+41-74574363
	Brian Kent	+1-347-589-7181
	David Smith	+33-1725625372
	James Poe	+49-795675134251

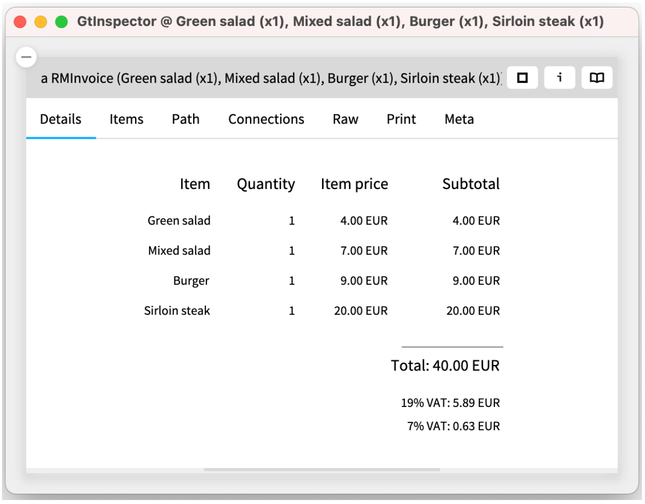
Figure 32: Domain-specific view for non-technical stakeholders

It is primarily the developer's responsibility to create *views* that are useful for other stakeholders. Any number of *views* can be attached to an object. Creating *views* does not require much effort, *e.g.*, on an average 12 lines of code for a view in the Glamorous toolkit. In Figure 55, we show two representations of an object of type `Invoice`. Figure 33a shows a raw view like in any other IDE, whereas Figure 33b shows a domain-specific custom representation for an invoice object proposed by a developer. Within citizen requirements, developers create *views* for the artifact as well as business objects. We believe that such custom and domain-specific graphical representations available natively in an IDE, are an excellent means for non-technical stakeholders to perform a variety of operations.


GtInspector @ Green salad (x1), Mixed salad (x1), Burger (x1), Sirloin steak (x1)
a RMInvoice (Green salad (x1), Mixed salad (x1), Burger (x1), Sirloin steak (x1)
Details
Items
Path
Connections
Raw
Print
Meta

Icon	Variable	Value
C	self	Green salad (x1), Mixed salad (x1), Burger
Σ	timestamp	2021-10-08T08:26:38.059514+05:30
C	announcer	an Announcer
{ }	invoiceItems	an OrderedCollection [4 items] (Green sal
C	price	40.00 EUR

(a) A raw view for an invoice object


GtInspector @ Green salad (x1), Mixed salad (x1), Burger (x1), Sirloin steak (x1)
a RMInvoice (Green salad (x1), Mixed salad (x1), Burger (x1), Sirloin steak (x1)
Details
Items
Path
Connections
Raw
Print
Meta

Item	Quantity	Item price	Subtotal
Green salad	1	4.00 EUR	4.00 EUR
Mixed salad	1	7.00 EUR	7.00 EUR
Burger	1	9.00 EUR	9.00 EUR
Sirloin steak	1	20.00 EUR	20.00 EUR

Total: 40.00 EUR

19% VAT: 5.89 EUR

7% VAT: 0.63 EUR

(b) A custom view for an invoice object

Figure 33: Custom object representations

3.3 Discussion

There are various implications of citizen requirements. Below, we briefly discuss three main implications. In chapter 6, we provide more details on two of them.

3.3.1 Impact analysis

Changing requirements impact the budget of a project and also affect the implementation. As we reported in chapter 2, existing tools are specialized for specific RE activities, often focusing on a limited set of artifacts. This approach scatters artifacts across several tools, thereby making linking a tedious task. In practice, several linking mechanisms are used to link and track artifacts [65, 106, 38, 62, 6, 14, 99, 81, 128]. These include manual referencing with artifact IDs, and using attachments for containers. However, there are several difficulties in applying linking mechanisms: the effort and time needed to perform the linking, managing obsolete links, and a lack of clear guidelines to establish a reliable linking structure.

In contrast, we maintain artifacts as first-class entities directly in the IDE. No additional linking and tracing mechanisms are required as plain object-oriented relationships are sufficient to connect a range of artifacts with each other as well as the artifacts and the application source code. The bidirectional references between requirements and the source code can help us assess the impact of the changes. This effect is favorable to both developers and project managers: developers know which exact classes or methods to update, managers plan the subsequent development iterations with a more accurate estimate of the remaining workload. We do not claim that our approach requires less effort for artifact creation. So far, the effort needed in deciding various tools to be used in a project toolchain, figuring out integration among selected tools can be diverted to creating artifacts in an IDE.

3.3.2 Modeling

To support agile development, models must be simple and easy to modify and should provide rapid feedback [12]. Researchers in recent years have tried to adapt several existing modeling mechanisms and approaches to be useful in agile development. These mostly include proposing development environments that serve as language workbenches to create domain-specific modeling languages (DSMLs) in an agile manner. Several DSMLs [24, 29, 142, 23, 59, 86], particularly graphical ones with interactive capabilities, are perceived to be suitable for domain experts [59, 90, 151]. Other studies provide an overview of existing modeling tools [26]. The proposed tools that support code generation and reverse engineering capabilities to varying degrees are all UML-based. Such modeling approaches can be heavyweight for application domains that are not well understood and are prone to evolve. Model-driven modeling approaches require considerable education and training. The fixed set of UML notations and the available tooling make UML suboptimal for agile modeling [104]. Likewise, the design and creation of an effective DSML requires language expertise, and significant upfront effort before the actual application development begins. The usability of DSMLs has been questioned, and in the context of highly volatile domains, the DSML itself must evolve continuously [9, 59].

Views on software development, such as "programming is conceptual modeling," or more generally, "programming is modeling," open a window to approach modeling for agile practices from a fresh perspective [53, 92, 37]. Rather than relying on up-front investment in a dedicated domain-specific language or model compiler, we envision a platform in which domain entities can be incrementally identified, added, and described. Object-oriented designs were introduced to close the gap between modeling and the code. Classes and objects in object-oriented design resemble the real world concepts, which are understandable by various stakeholders. However, creating classes and run-time objects is largely seen as a task of technical stakeholders, while GUIs are desired to support non-technical stakeholders to manipulate the business objects. This is evident from various visual modeling approaches (*e.g.*, business process modeling) and tools (*e.g.*, Rational Rose). In chapter 6, we discuss how we can automate the object-oriented design process by providing custom GUIs and a workflow, and support non-technical stakeholders to follow the best practices of the object-oriented design. With citizen requirements, the process of identifying domain entities, describing their behavior, and specifying the relationships among them can be supported natively in an IDE.

3.3.3 Documentation

Project documentation is often created from existing artifacts, and to date, no dedicated tool guarantees to keep it up-to-date [79, 8]. Documentation is vital for future development, and must be created iteratively and collaboratively, and kept up-to-date by all the team members [123]. Difficulties with linking can leave artifacts out-of-sync, and hurt project documentation. As far as IDEs are concerned, Visual Studio Code, for example, supports Markdown editing, and as such, it is useful for documentation. Markdown format is less suitable for embedding a range of custom created artifacts. Another example is Jupyter Notebook, which supports writing code and documentation within the same tool. However, Jupyter Notebook is primarily used for data engineering, and achieving an executable and interactive documentation for a working software, to the best of our knowledge is not yet explored with Jupyter Notebook [143].

On the contrary, being first-class entities, artifacts with our approach are readily available in the IDE to prepare documentation. As artifacts serve distinct purposes, we can use them to create specific types of documents. With our approach, documents become just another artifact that leverages other artifacts to present something meaningful to a user. For example, scenarios are used to test a software feature against a number of input conditions. They can be used to prepare a document that serves as a tutorial to explain a specific software feature. In chapter 6, we revisit the idea of live programming, and envision a live programming environment where changes made to artifacts are immediately visible, giving stakeholders rapid feedback, much required in agile development.

3.4 Conclusion

In this chapter, we revisited three existing research ideas, which we leverage to propose an approach to support various stakeholders in agile development. We presented the vision and three building blocks of the citizen requirements approach to support agile development within an IDE. With these building blocks, we enable non-technical stakeholders to participate in creating and maintaining requirements directly in an IDE. Likewise, they can interactively create test cases to verify the application's behavior. Finally, we briefly discussed some of the implications of our proposal.

Chapter 4

Collaborative specification and management of requirements

A multitude of artifacts are used for distinct tasks in software development, as well as in requirements engineering [81, 57], each with its own merits and limitations [81, 114]. There are, for example, design artifacts [40, 30], requirements artifacts [60], agile modeling artifacts [12], and software artifacts [75]. We stick to a broader definition of an artifact, which considers any object that is created and used to facilitate work activities relating to a software project [56, 148]. To boost artifact connectivity and maintain their consistency, we propose to create and manage software-related artifacts as first-class entities directly in an IDE. This approach has two advantages: (i) compared to employing separate tools, creating various artifacts directly within a development platform eliminates the necessity to recover trace links, and (ii) first-class artifacts can be composed into stakeholder-specific live documentation. In this chapter, we detail and exemplify this idea using the Glamorous toolkit. We present three artifacts, namely user stories, mindmaps, and scenarios, to demonstrate that our approach is feasible. Note that we do not discuss in detail how artifacts are built from scratch, but we show how they can be used in a specific situation. The construction of artifacts from scratch can vary from artifact to artifact and require different amounts of effort. The results of this study have been accepted for publication at the 29th IEEE International Conference on Software Analysis, Evolution and Re-engineering (SANER).

4.1 Collaborative artifacts building

4.1.1 A running example

We learned from a medical doctor how inefficiently their hospital prepares its roster, thereby leaving no chance for doctors to have any social life. We use it as an example to exemplify the first part of the citizen requirements approach as it resembles a real-word situation and is elaborate enough to discuss various artifacts.

Suppose a hospital needs to prepare its roster efficiently, and its management wants to update or replace its existing shift scheduling software. Following agile development practice, a business analyst needs to discuss requirements with domain experts from the hospital (who are also the business stakeholders here), express the requirements in some format, and then update the scheduling software system in use. After every development iteration, the developers need to present new functionality to the business stakeholders. After each such meeting, the development team gets feedback and proceeds to update the requirements, which means they need to update various artifacts, change the implementation, and update the documentation respectively. Preparing a schedule is a tedious task as the staff member responsible for preparing a schedule needs to take into account numerous constraints, such as those related to permissible working hours, constraints for assigning medical staff to each shift, *etc.*

4.1.2 Workflow

The workflow below can be applied to build any artifact one wishes to model in an IDE. Developers are required to invest once in building artifacts from scratch. Once such infrastructure exists, it can be used for any future project.

Create meta infrastructure: artifact classes

First, developers need to build the requirements-related artifacts (*e.g.*, user stories, mindmaps) of choice as first-class entities from scratch, *i.e.*, by creating appropriate classes and implement their behavior in the methods. For instance, a user story might be modeled with a class `UserStory` and have specific behavior implemented in a method `setStatus` that will enable users to set the status of a particular user story object. Developers also specify relationships among them. For example, a Kanban board used to track work progress in agile practices can be modeled to host user stories. Likewise, the `UserStory` class can be modeled to have references to source code classes that model the domain entities referred to in a specific user story. For example, concepts, such as waiter or order, from a user story: *As a waiter, I want to add Pizza and Cappuccino to an order*, can be connected to the corresponding `Waiter` and `Order` classes.

Create GUIs

Once various artifact classes are created and their relationships explicitly modeled, developers then build custom GUIs, which enable other stakeholders to create, access, and navigate the corresponding artifacts. For instance, a project manager can create project-specific user stories as first-class entities by using an in-IDE graphical interface instead of using dedicated tools, such as Trello. Developers create appropriate *views* for the artifacts. For example, a user story object can be represented using a story card metaphor for easy and intuitive manipulation.

4.2 Example artifacts

Next, we discuss three artifacts that support their users in distinct software development tasks. They are created in one phase and are used in other phases of the development lifecycle. The implementation of the running example and the following artifacts can be explored by following the instructions provided in the readme file.[1] Specifically, we showcase artifacts used in:

- requirements engineering: user stories;
- modeling: a mindmap;
- testing: scenarios.

4.2.1 User stories

To record requirements in a collaborative way, there is a need for an artifact that can be conveniently edited by technical and non-technical stakeholders alike and is lightweight to manage. User stories are artifacts that serve to record requirements from the end-user perspective [82]. User stories are a nice fit for the IT company to collect and specify requirements together with the hospital staff. In Figure 41, we show two representations of user stories. The “Raw” representation shows raw data about a user story object, while the “Minimal” representation of the same user story object presented as a card gives additional details, such as assigned labels and team members, of a specific user story, which are typically needed by project managers. A user story object, being a first-class entity, can be embedded anywhere, in any live document, or into a live Kanban board.

4.2.2 Mindmaps

Mindmapping is a visual way of organizing and representing information within a radial hierarchy [83]. The most important concept appears at the center of a given diagram and related concepts are connected via edges. Based on their relevance, the related concepts appear farther and farther away from the center of the diagram. Let us consider that a new developer joins the development team and wants to understand the hospital

[1] `https://github.com/nitishspatkar/moldable-requirements`

management domain. A mindmap of domain concepts from the scheduling application could assist a new developer in understanding the main concepts.

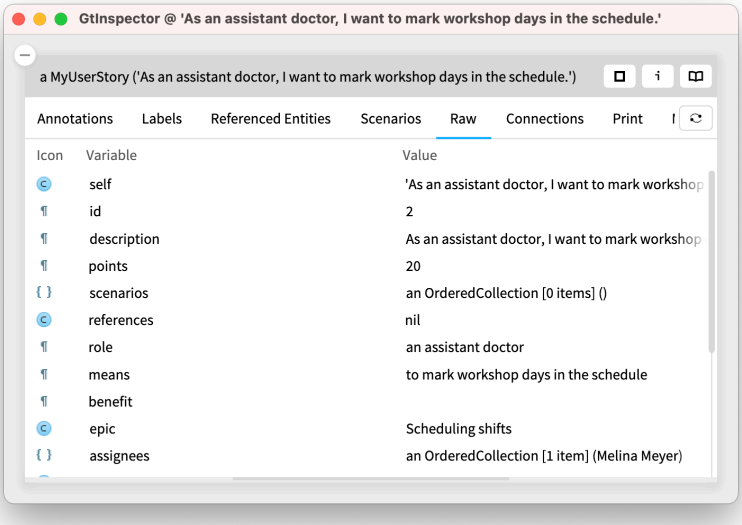
In Figure 42, we show a mindmap of main domain entities in our scheduling application. In Figure 42a, a user has clicked on a node "HMDoctor," and an object inspector window on the right-hand side shows the class comment for the "HMDoctor" class, which allows the new developer to understand the implementation details of each domain concept in an iterative and interactive manner. Likewise, in Figure 42b, we show the tab "Related Stories," which enables a developer to explore the related user stories (*i.e.*, requirements) for a specific domain concept. Such a binding mechanism fosters two-way connectivity between two artifacts.

4.2.3 Scenarios

Scenarios are popular in practice as they exhibit potential for collaborative construction and review. Unlike test cases, a scenario contains high-level documentation, which describes an end-to-end functionality to be tested. Scenarios are created in various formats. For example, a UML sequence diagram models a specific interaction scenario. Likewise, BDD scenarios are written using the Gherkin language and as discussed earlier, face issues with redundancy. Let us consider that we want to see how our implementation of the scheduling algorithm assigns a fixed number of doctors to one day, to a week, and a month. In Figure 43, we show an executable scenario that prepares a schedule for one week for the available medical staff. This piece of source code serves as a test case that asserts a condition and returns the corresponding object. In the leftmost window, we see for a specific class, all the related scenarios collected at one place under the "Related scenarios" tab. In our example, we want to observe the resulting hospital management system with a seven day schedule. The middle window is an object inspector on an object of type `HospitalSystem` that lists the seven days, and when a user clicks on a specific day, the corresponding schedule for that day can be explored in another object inspector. By executing different scenarios, a user explores how a system behaves under different conditions. Note that such scenarios can be fully generated from an in-IDE user interface, see chapter 5 for more details.

4.3 Evaluation

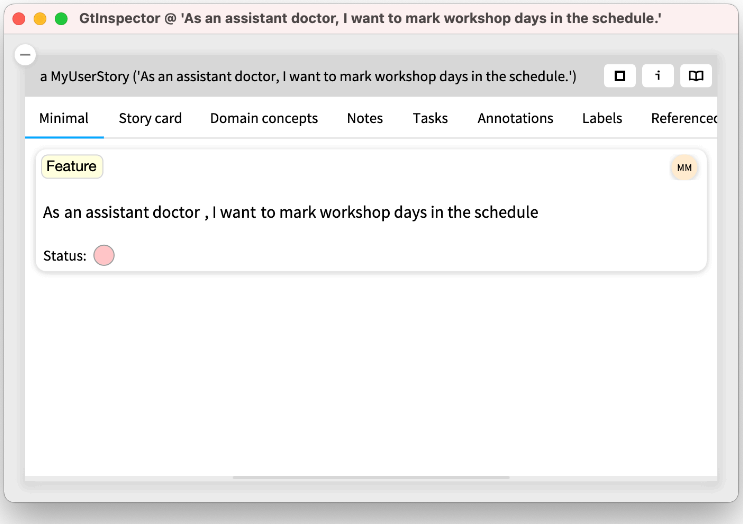
To obtain some early feedback on the potential of our idea to be beneficial for artifact management and, in particular, being suitable for non-technical stakeholders, we conducted a semi-structured pilot survey with three practitioners and researchers. We selected the participants through mutual contacts. The participants had varying experiences with software development and agile methodologies, ranging within 7-20 years. The on-


GtInspector @ 'As an assistant doctor, I want to mark workshop days in the schedule.'

a MyUserStory ('As an assistant doctor, I want to mark workshop days in the schedule.')

Icon	Variable	Value
C	self	'As an assistant doctor, I want to mark workshop
¶	id	2
¶	description	As an assistant doctor, I want to mark workshop
¶	points	20
{ }	scenarios	an OrderedCollection [0 items] ()
C	references	nil
¶	role	an assistant doctor
¶	means	to mark workshop days in the schedule
¶	benefit	
C	epic	Scheduling shifts
{ }	assignees	an OrderedCollection [1 item] (Melina Meyer)

(a) A raw view for a user story


GtInspector @ 'As an assistant doctor, I want to mark workshop days in the schedule.'

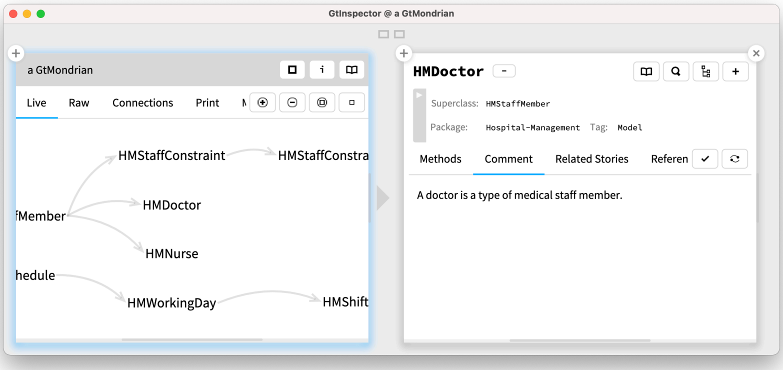
a MyUserStory ('As an assistant doctor, I want to mark workshop days in the schedule.')

As an assistant doctor , I want to mark workshop days in the schedule

Status:

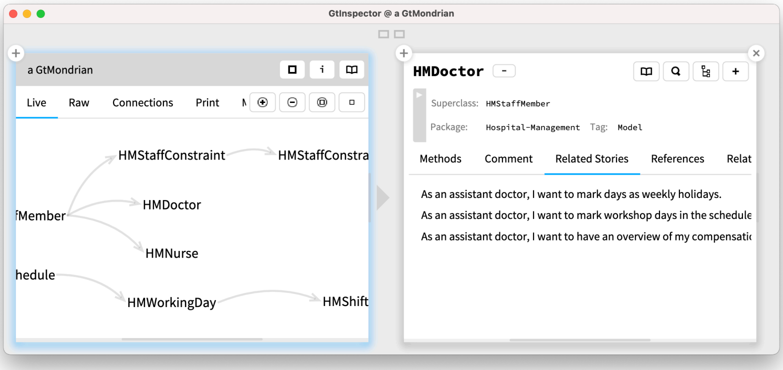
(b) A minimal card view for a user story

Figure 41: Custom entity representations



A doctor is a type of medical staff member.

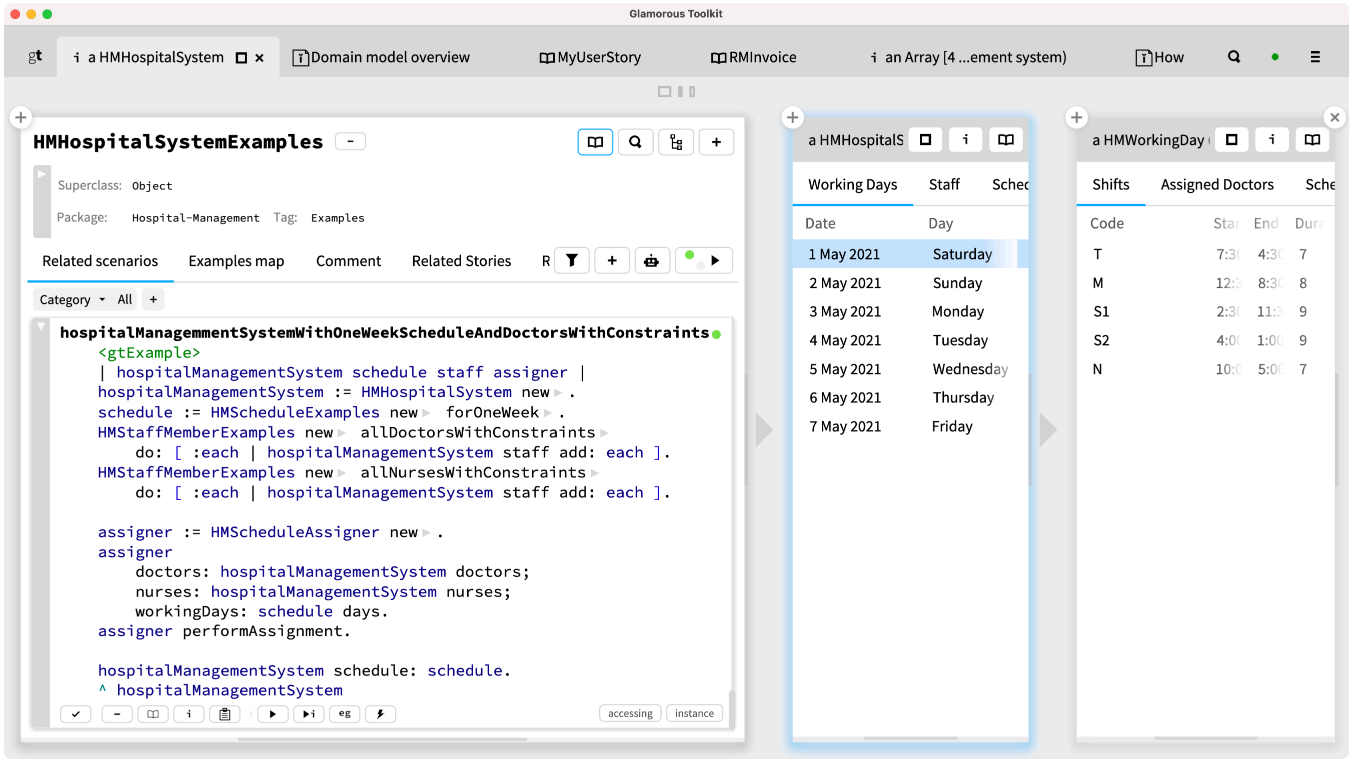
(a) Exploring class comments for a domain concept



As an assistant doctor, I want to mark days as weekly holidays.
As an assistant doctor, I want to mark workshop days in the schedule
As an assistant doctor, I want to have an overview of my compensatic

(b) Exploring user stories related to a domain concepts

Figure 42: A sample in-IDE mindmap



```
hospitalManagemmentSystemWithOneWeekScheduleAndDoctorsWithConstraints
    <gtExample>
    | hospitalManagementSystem schedule staff assigner |
    hospitalManagementSystem := HMHospitalSystem new .
    schedule := HMScheduleExamples new forOneWeek .
    HMStaffMemberExamples new allDoctorsWithConstraints
        do: [ :each | hospitalManagementSystem staff add: each ].
    HMStaffMemberExamples new allNursesWithConstraints
        do: [ :each | hospitalManagementSystem staff add: each ].

    assigner := HMScheduleAssigner new .
    assigner
        doctors: hospitalManagementSystem doctors;
        nurses: hospitalManagementSystem nurses;
        workingDays: schedule days.
    assigner performAssignment.

    hospitalManagementSystem schedule: schedule.
    ^ hospitalManagementSystem
```

Working Days | Staff | Schec

Date	Day
1 May 2021	Saturday
2 May 2021	Sunday
3 May 2021	Monday
4 May 2021	Tuesday
5 May 2021	Wednesday
6 May 2021	Thursday
7 May 2021	Friday

Shifts | Assigned Doctors | Sche

Code	Star	End	Dura
T	7:3	4:3	7
M	12:3	8:3	8
S1	2:3	11:3	9
S2	4:0	1:00	9
N	10:0	5:0	7

Figure 43: A sample in-IDE executable scenario

line survey consisted of the following steps: a brief introduction to the identified issues with artifact management, an introduction to the proposed approach, followed by a short 15 minute demo, and finally, an online survey for the participants. The survey instrument was prepared and validated by the study team collaboratively, and consisted of questions regarding participants' background and their feedback on various aspects of the proposed approach. We have included the survey instrument and the responses in the provided additional supporting material, in section A.5. We conducted a pilot demo session to ensure timely execution. The results, in particular the answers to the open-ended questions, were codified by the study team.

All of the participants agreed that our approach could help project teams in managing artifacts and handling artifact traceability, and will reduce the number of tools employed in a software project. Similarly, all participants agreed that our approach could reduce the context switches between various tools to accomplish a single development-related task and provide more accurate matrices (*e.g.*, pending workload) for decision making. Notably, all of the participants strongly agreed that our approach could reduce the manual effort required in keeping the project documentation up-to-date.

While reflecting on the perceived advantages, they mentioned that "... *the approach can help to build a common language among different roles in a project and could work as a single source of truth.*" Similarly, it could also help with "... *better onboarding of new team members*," "... *better understanding of relations between different artifacts*," and "... *communication between domain and technical experts.*" Other stated advantages of our approach were that it leads to "... *up-to-date living documentation*," and "... *shared understanding*" among several stakeholders. The notable limitations mentioned by participants concerned the usability of the graphical interfaces presented in the demo and the approach's scalability in a large-scale project. One participant also expressed concern regarding the effort that one might require to integrate numerous artifacts of different types.

4.4 Threats to the validity

- Less number of participants
- No real customers involved

4.5 Conclusion

To avoid scattering of various software-related artifacts among separate tools, we argued that artifacts should be created as first-class entities directly in an IDE. Our proposed approach helps maintain various software-related artifacts in one platform, eliminating a need to recover trace links. Overall, this approach simplifies the artifact management and involves various stakeholders in the development process equally. Moreover, with up-to-date artifacts, various stakeholders can provide quick feedback to others much desired in agile development. We presented an advanced prototype implementation of three artifacts in the Glamorous toolkit. We also conducted a semi-structured online pilot survey with practitioners and researchers to evaluate the potential of our idea for artifact management. The initial results are encouraging for us to continue with this research line.

Chapter 5

Collaborative requirements validation

In the previous chapter, we discussed how requirements can be created as first-class artifacts directly in an IDE. As we discussed in section 2.3, the current workflow support for BDD expects non-technical stakeholders to use an IDE to specify textual scenarios in the Gherkin language and verify the behavior using test passed/failed reports. Research to date shows that this approach leads them to write redundant Gherkin specifications and makes testers perceive BDD as an overhead in addition to the testing. In this chapter, we discuss how we can engage non-technical stakeholders in verifying requirements in an IDE. The results of this study have been published in the Proceedings of the 25th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings.

5.1 Collaborative validation

5.1.1 A running example

To exemplify the second part of citizen requirements, we need an example that resembles a real-world situation. When the Corona pandemic started in Europe and lockdowns were imposed in various parts in March 2020, the German federal government proposed changes to the value-added tax (VAT) system to help the gastronomy industry.[1] We find the new VAT rules quite interesting and wonder how they can be implemented on a short notice in all restaurants.

Let us consider that we need to update an existing invoicing system for a restaurant and consequently verify if the new invoices are calculated correctly. The invoicing system allows its users to add menu items to an order, and indicate if those items were consumed inside the restaurant

[1]The new VAT system: accessed November 12, 2020, https://www.hellotax.com/blog/new-vat-rates-germany/

or were ordered for take away. The new invoices should correctly reflect a change in value added tax (VAT) calculation. A different VAT should apply for the same menu item depending on whether it is for take away or on-site consumption.

Menu item	On site VAT	Takeaway VAT %
Black coffee	19	19
Cappuccino	19	7
Pizza Margherita	19	7

The invoice contains the total cost of ordered menu items and a VAT. The invoicing example is a little more complicated as it involves complex domain objects, such as of type `Invoice` and `Order`. To verify the user story *As a waiter, I want to be able to view the total price before printing the invoice*, one can write scenarios such as:

```
Scenario 1: Customers place an order to take away
(only milk products)
Given an empty order
When the waiter adds Cappuccino to the empty order
And a cup of Cappuccino costs 4 EUR
And a cup of Cappuccino is taxed at 7%
And the waiter generates the Invoice for the order
Then the total invoice price is 7.28 EUR

Scenario 2: Customers place an order to take away
(combination of non-milk and milk products)
Given an empty order
When the waiter adds a Cappuccino and a black coffee
to the empty order
And a cup of Cappuccino costs 4 EUR
And a cup of black coffee costs 3 EUR
And a cup of Cappuccino is taxed at 7%
And a cup of black coffee is taxed at 19%
And the waiter generates the Invoice for the order
Then the total invoice price is 7.85 EUR
```

Listing 8: Sample scenarios for the invoicing application

The final price in the *Then* statement in the first scenario is purposefully incorrect. In section A.6, we list the complete requirements for such an application decomposed into user stories, and corresponding scenarios.

The typical current BDD workflow faces two issues here. First, this workflow leads non-technical stakeholders to write numerous scenarios with minor variations, such as in input parameter values, and requires them to specify the test assertions. As we mentioned earlier, keywords, such as *Scenario Outline*, and data tables were introduced to reduce the redundancy in Gherkin specification. However, our manual inspection of 23 open-source projects shows that data tables are rather moderately used. Second, using the test run status as a means to verify behavior obscures details of logical mistakes made in the scenario specification. A non-technical stakeholder manually had to calculate the expected results during specification. Although it is a common practice in testing in general, it can lead to software run-time errors that are difficult to locate in the textual specifications.

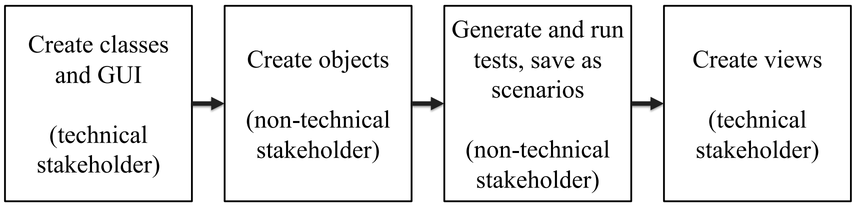

Create classes and GUI
(technical stakeholder)
Create objects
(non-technical stakeholder)
Generate and run tests, save as scenarios
(non-technical stakeholder)
Create views
(technical stakeholder)

Figure 51: Proposed BDD process

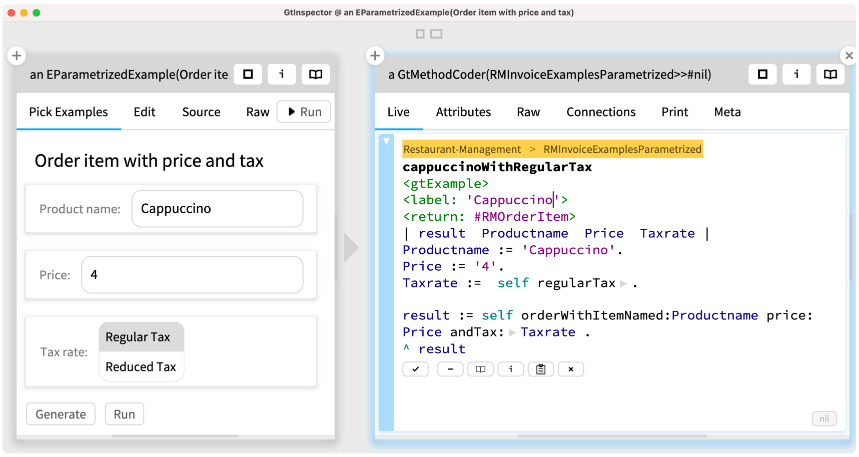
5.1.2 Workflow

In Figure 51, we outline our proposed BDD workflow. Developers will create classes and implement the behavior of those classes in the methods. However, instead of specifying behavior and updating test cases, they only need to insert the assertions in the fully-generated test cases. Developers will also create GUIs for object creation and views to explore details of run-time objects visually. Non-technical stakeholders, on the other hand, will use GUIs to compose and save scenarios. If they wish, they can also insert the assertions in the fully-generated test cases. In other words, non-technical stakeholders do not need to write textual scenarios. Instead of test run status, they will use a domain-specific representation of the involved objects to verify the implemented behavior. Technical stakeholders, on the other hand, need to implement GUIs for object creation and object representation.

Let us consider that we have two stakeholders, Bob, who is a non-technical domain expert, and Melinda, the developer. As scenarios elaborate a specific user story, by using the building blocks (*i.e.*, GUIs, examples, and views) Bob can interactively create scenarios as first-class entities as described next. Note that except for the test case generation, no other step in the workflow is automated– each step still requires manual effort from the concerned stakeholders, however the type of interaction with the system is different compared to the existing BDD workflow. Our proposed workflow divides the process of scenario creation and verification into the following four steps: (1) create classes and GUIs, (2) create domain objects, (3) generate test cases and save as scenarios, and (4) create *views*. We detail below each step.

Create meta-infrastructure: classes and GUIs

The user story *As a waiter, I want to add menu items to prepare an order* contains the following domain concepts: `Waiter`, `MenuItem`, and `Order`. This user story also specifies expected behavior, *i.e.*, an order is created by adding menu items to it. Melinda creates classes for the domain concepts involved in a particular user story and builds GUI that would enable Bob to create run-time objects for these classes. For instance, the GUI in Fig-



GtInspector @ an EParametrizedExample(Order item with price and tax)

an EParametrizedExample(Order ite

Order item with price and tax

Product name: Cappuccino

Price: 4

Tax rate: Regular Tax
Reduced Tax

a GtMethodCoder(RMInvoiceExamplesParametrized>>#nil)

```
Restaurant-Management > RMInvoiceExamplesParametrized
cappuccinoWithRegularTax
<gtExample>
<label: 'Cappuccino'>
<return: #RMOrderItem>
| result  Productname  Price  Taxrate |
Productname := 'Cappuccino'.
Price := '4'.
Taxrate :=  self regularTax .

result := self orderWithItemNamed:Productname price:
Price andTax: Taxrate .
^ result
```

Figure 52: A GUI to create a simple domain object and save the example as an operation

ure 52 enables creating run-time objects of type `MenuItem`. This interface enables Bob to create a `Cappuccino` object, and provide details, such as the price and VAT. Melinda also implements the methods that define behavior for each class. For generic GUIs, which are often standard *views* to show a list of something, the cost of creation is similar as in any other language. However, the GUIs for composing *examples* are specific ones, which might require editing of properties of objects. The creation of such kinds of GUIs could also be optimized. One way to do it is using existing frameworks, such as Magritte.[2] Magritte needs an object with some annotations, and from that model it generates GUIs for editing that object [109]. In our case, we generate GUIs for editing the *examples* from the *example method*, see Figure 53. The *view* uses the annotations in the source code of the example to determine how to create the GUI for specifying parameters for the *example*.

Create domain objects

Now, Bob can create several instances of concerned classes by using the provided GUI. When he clicks on the "Generate" button in Figure 52, the corresponding example method is created, see the right hand window. When anyone executes this newly created example method, it always returns the same `Cappuccino` object with selected price and tax. Once the basic domain objects are created, they can be used to create more complex domain objects. For example, an `Order` could be composed from various existing `MenuItems`. To enable Bob to create such complex domain objects,

[2] "Magritte," https://github.com/magritte-metamodel/magritte

Melinda creates a tailored graphical interface. For instance, the graphical interface in Figure 54 is populated with various already created simple objects, *i.e.*, `Cappuccino` and `Coffee`, that appear as a list in a drop-down menu. Bob uses this interface to create complex objects and save the selection as another example method (see Figure 55a). Here, Bob creates an `Invoice` object for an `Order` with two menu items (*i.e.*, `Cappuccino` and `Coffee`). However, instead of "Generate," now he clicks on the "Run" button to explore the resulting `Invoice` object visually.

Create test cases and save scenarios

The tailored graphical interfaces, shown in Figure 52 and Figure 54, essentially enable Bob to create both simple and complex domain objects and also generate an example method that when executed returns a specific domain object. Examples represent a concrete scenario. To save an example method as a scenario, Bob clicks on the "✓" button in the right hand side window of Figure 52. This saves the newly created scenario for a particular user story, and Bob can always access it from one of the views for a `UserStory` (see Figure 56). Melinda or Bob add assertions (see Figure 57) to this newly generated example method to test the specified behavior in the respective methods of the domain classes. With this approach, instead of writing scenarios textually, Bob could interactively create simple domain objects (*e.g.*, `Cappuccino` and `Coffee`) and use those to create complex domain objects (*e.g.*, `Order`). He could save the selection of `Cappuccino` and `Coffee` to an `Order` as an example method, which will return the same `Order` instance with `Cappuccino` and `Coffee` when executed. This example method is attached to the **`Scenario`** object.

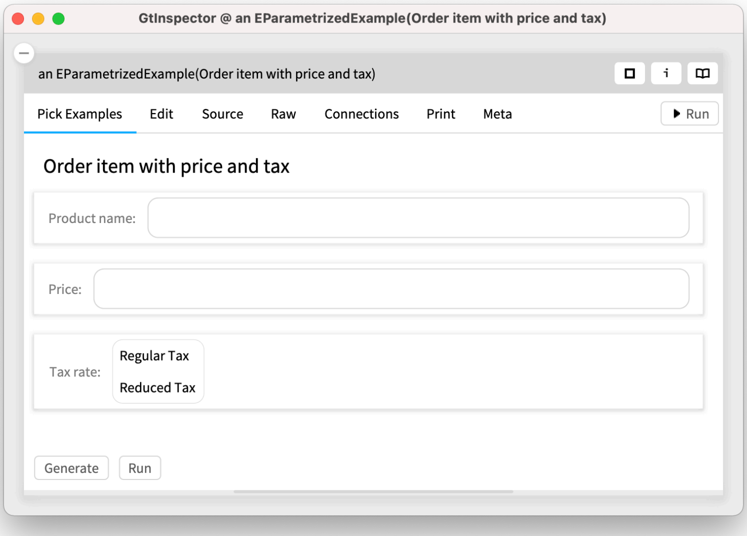
Create views

Both Bob and Melinda need different representations of domain objects to accomplish distinct tasks. For instance, Bob, being a non-technical stakeholder, needs to determine whether the correct number of menu items are added to an `Order` object, whether correct prices and tax rates are applied to each `MenuItem` object, and whether the final price is accurately calculated in `Invoice` object. He uses the printable representation of the `Invoice` object in Figure 55a that fulfills his needs. Likewise, Melinda, being a developer, needs to understand how an `Invoice` object is constructed. She uses the composition presentation of the `Invoice` object in Figure 55b to explore how it is made up of other objects, such as of type `Cappuccino`, with their corresponding properties, such as applied tax rates. Note that the process of creating objects and *views* is iterative and incremental—*views* can be designed as the necessity arises to explore some specific details of a specific domain object. Theoretically, Melinda could create the printable invoice view when she first created the class `Invoice`.

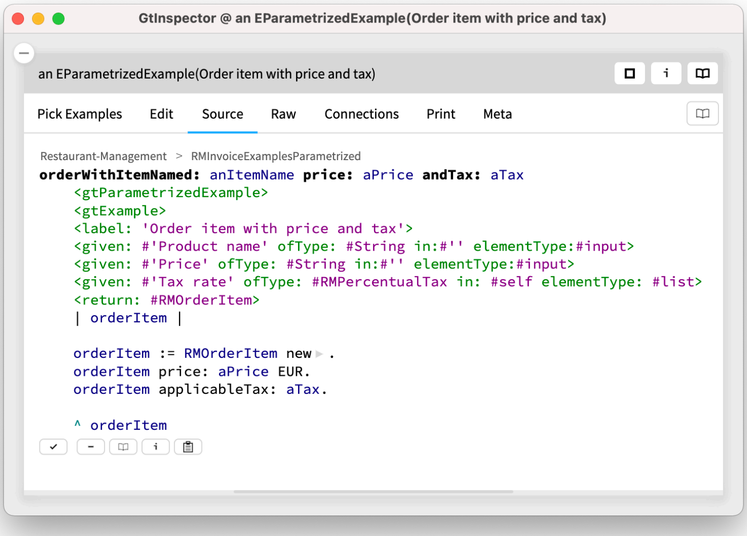
With this approach the application behavior becomes verifiable by stakeholders by inspecting domain objects instead of reading a test report. Notably, this approach does not eliminate the need for test cases. The example methods not only serve as test cases, but also augment them with domain-specific representations of the involved run-time objects.

5.2 Conclusion

In this chapter, we proposed an alternative BDD process to engage both technical and non-technical stakeholders in specifying and verifying the application behavior directly in an IDE. We demonstrate through a running example of invoicing system for restaurants how non-technical stakeholders can visually compose behavior tests and discover inconsistencies in the underlying domain model through an inspectable output. Our proposed building blocks for an IDE allow to better integrate both technical and non-technical stakeholders in the BDD process.


GtInspector @ an EParametrizedExample(Order item with price and tax)
an EParametrizedExample(Order item with price and tax)
Pick Examples
Edit
Source
Raw
Connections
Print
Meta
Run
Order item with price and tax
Product name:
Price:
Tax rate:
Regular Tax
Reduced Tax
Generate
Run

(a) A GUI to create a simple domain object


GtInspector @ an EParametrizedExample(Order item with price and tax)
an EParametrizedExample(Order item with price and tax)
Pick Examples
Edit
Source
Raw
Connections
Print
Meta
Restaurant-Management > RMInvoiceExamplesParametrized

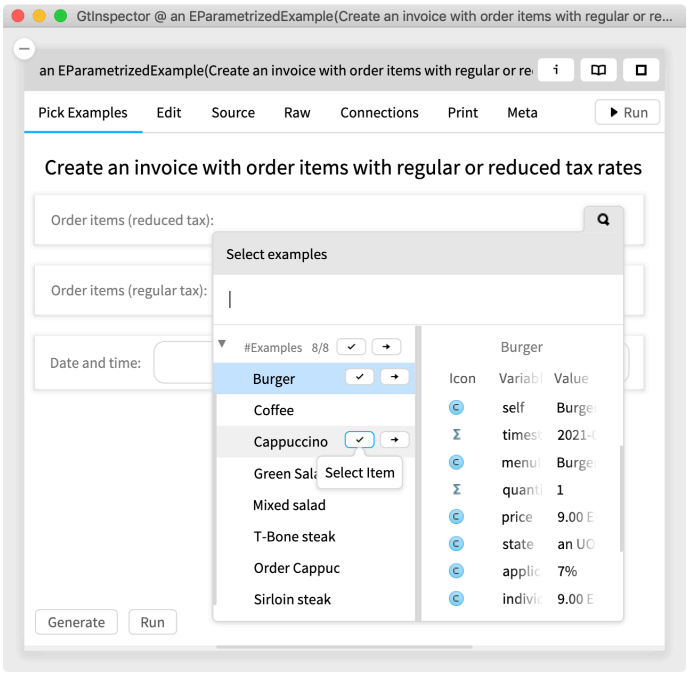
```
orderWithItemNamed: anItemName price: aPrice andTax: aTax
    <gtParametrizedExample>
    <gtExample>
    <label: 'Order item with price and tax'>
    <given: #'Product name' ofType: #String in:#'' elementType:#input>
    <given: #'Price' ofType: #String in:#'' elementType:#input>
    <given: #'Tax rate' ofType: #RMPercentualTax in: #self elementType: #list>
    <return: #RMOrderItem>
    | orderItem |

    orderItem := RMOrderItem new .
    orderItem price: aPrice EUR.
    orderItem applicableTax: aTax.

    ^ orderItem
```

(b) The source code of the corresponding *example* method

Figure 53: Creating a custom GUI


GtInspector @ an EParametrizedExample(Create an invoice with order items with regular or re...
an EParametrizedExample(Create an invoice with order items with regular or re
Pick Examples
Edit
Source
Raw
Connections
Print
Meta
Run
Select examples
#Examples 8/8
Select Item
Generate
Run

Create an invoice with order items with regular or reduced tax rates

Order items (reduced tax):

Order items (regular tax):

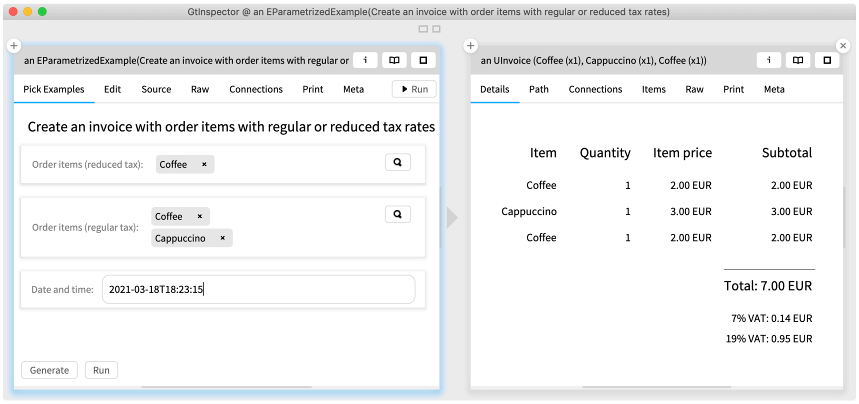
Date and time:

Burger
Coffee
Cappuccino
Green Sal
Mixed salad
T-Bone steak
Order Cappuc
Sirloin steak

Burger

Icon	Variabl	Value
C	self	Burge
Σ	timest	2021-
C	menu	Burge
Σ	quanti	1
C	price	9.00 E
C	state	an UO
C	applic	7%
C	indivic	9.00 E

Figure 54: A GUI to select simple domain objects


GtInspector @ an EParametrizedExample(Create an invoice with order items with regular or reduced tax rates)

an EParametrizedExample(Create an invoice with order items with regular or

Create an invoice with order items with regular or reduced tax rates

Order items (reduced tax): Coffee

Order items (regular tax): Coffee, Cappuccino

Date and time: 2021-03-18T18:23:15

an UInvoice (Coffee (x1), Cappuccino (x1), Coffee (x1))

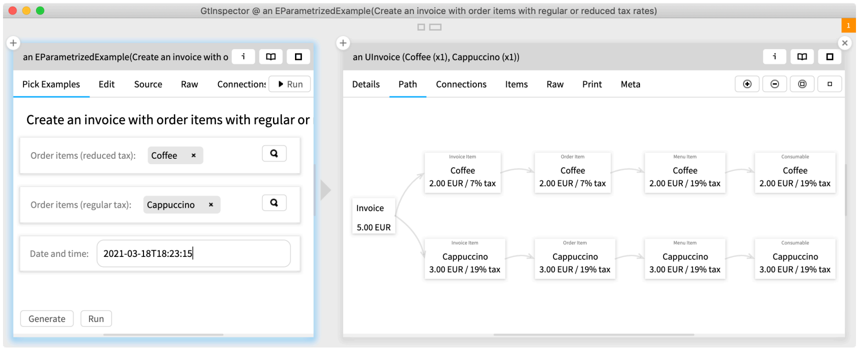
Item	Quantity	Item price	Subtotal
Coffee	1	2.00 EUR	2.00 EUR
Cappuccino	1	3.00 EUR	3.00 EUR
Coffee	1	2.00 EUR	2.00 EUR

Total: 7.00 EUR

7% VAT: 0.14 EUR

19% VAT: 0.95 EUR

(a) A GUI to create complex domain object and explore the resulting object with printable view


GtInspector @ an EParametrizedExample(Create an invoice with order items with regular or reduced tax rates)

an EParametrizedExample(Create an invoice with o

Create an invoice with order items with regular or

Order items (reduced tax): Coffee

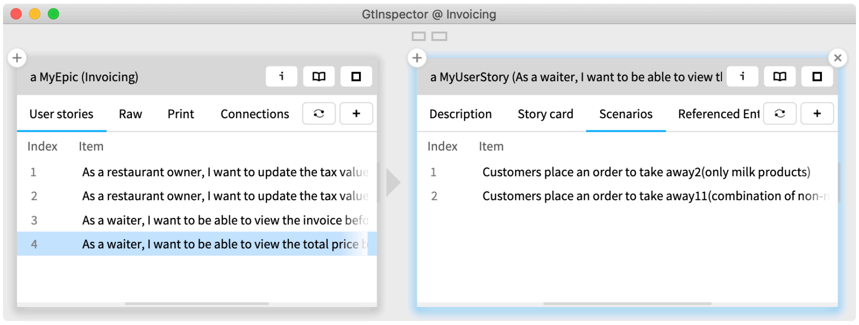
Order items (regular tax): Cappuccino

Date and time: 2021-03-18T18:23:15

an UInvoice (Coffee (x1), Cappuccino (x1))

(b) A GUI to create complex domain object and explore the resulting object with composition view

Figure 55: Custom entity representations


GtInspector @ Invoicing

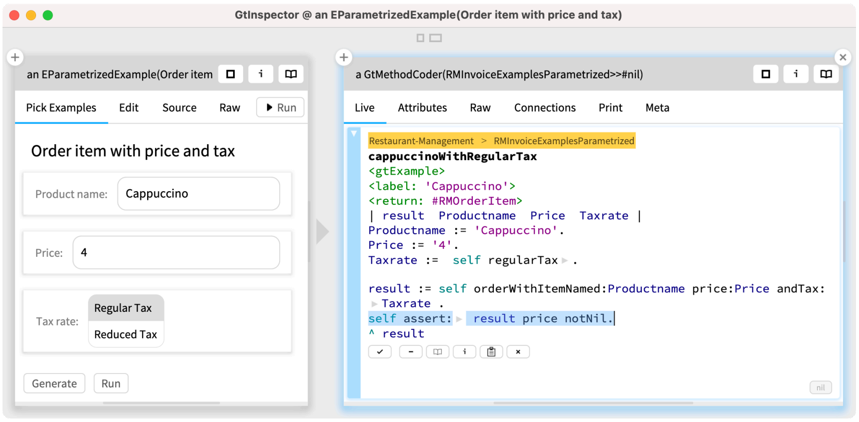
a MyEpic (Invoicing)

Index	Item
1	As a restaurant owner, I want to update the tax valu
2	As a restaurant owner, I want to update the tax valu
3	As a waiter, I want to be able to view the invoice bef
4	As a waiter, I want to be able to view the total price

a MyUserStory (As a waiter, I want to be able to view tl

Index	Item
1	Customers place an order to take away2(only milk products)
2	Customers place an order to take away11(combination of non-

Figure 56: Scenarios are attached to a user story


GtInspector @ an EParametrizedExample(Order item with price and tax)
an EParametrizedExample(Order item
Pick Examples
Edit
Source
Raw
Run
a GtMethodCoder(RMInvoiceExamplesParametrized>>#nil)
Live
Attributes
Raw
Connections
Print
Meta
Generate
Run
nil

Order item with price and tax

Product name: Cappuccino

Price: 4

Tax rate: Regular Tax / Reduced Tax

Restaurant-Management > RMInvoiceExamplesParametrized

```
cappuccinoWithRegularTax
<gtExample>
<label: 'Cappuccino'>
<return: #RMOrderItem>
| result  Productname  Price  Taxrate |
Productname := 'Cappuccino'.
Price := '4'.
Taxrate :=  self regularTax .

result := self orderWithItemNamed:Productname price:Price andTax:
 Taxrate .
self assert: result price notNil.
^ result
```

Figure 57: Adding assertions to an example method

Chapter 6

Implications

In this chapter, we discuss two main implications of *citizen requirements*, namely, live documentation and agile domain modeling.

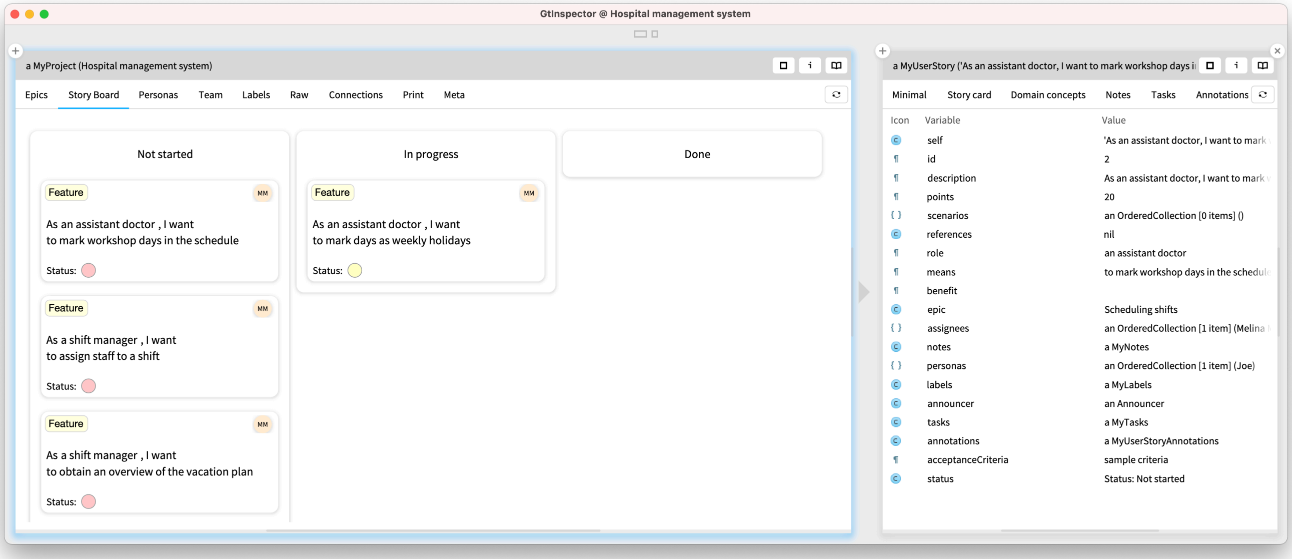
6.1 Live documentation

Live programming environments historically provide immediate feedback to developers about their changes to the running system. Early research in this area was mostly limited to constructing live visual languages [126, 127, 28]. More recent research efforts in live programming support live data analysis and visualization [77, 46, 71, 150].

With the same idea discussed in chapter 4, we can create various project-related documents as first-class entities themselves. One can dynamically create several documents that consume different artifacts and explain specific aspects of a running system. For non-technical stakeholders such documents can aggregate project-related information for requirements-related entities. In this section, we show how we can compose user stories, mindmaps, and scenarios into an interactive and live documentation. We explain using two situations and a type of documentation that might help a specific stakeholder to accomplish a goal in that situation.

6.1.1 A Kanban board

Now, let us consider a non-technical stakeholder, such as a product owner, needs an overview of the up-to-date progress of the project. Traditionally, she will use an existing project management tool, such as Trello or GitHub projects. Such tools help to track a lot of development progress-related data: an overview of accomplished work, types of remaining tasks in the pipeline *etc.* With our approach, we can provide a similar overview in an IDE itself.


GtInspector @ Hospital management system
a MyProject (Hospital management system)
Epics
Story Board
Personas
Team
Labels
Raw
Connections
Print
Meta

Not started

Feature MM
As an assistant doctor , I want
to mark workshop days in the schedule
Status:

Feature MM
As a shift manager , I want
to assign staff to a shift
Status:

Feature MM
As a shift manager , I want
to obtain an overview of the vacation plan
Status:

In progress

Feature MM
As an assistant doctor , I want
to mark days as weekly holidays
Status:

Done

a MyUserStory ('As an assistant doctor, I want to mark workshop days i
Minimal
Story card
Domain concepts
Notes
Tasks
Annotations

Icon	Variable	Value
	self	'As an assistant doctor, I want to mark
	id	2
	description	As an assistant doctor, I want to mark
	points	20
	scenarios	an OrderedCollection [0 items] ()
	references	nil
	role	an assistant doctor
	means	to mark workshop days in the schedule
	benefit	
	epic	Scheduling shifts
	assignees	an OrderedCollection [1 item] (Melina
	notes	a MyNotes
	personas	an OrderedCollection [1 item] (Joe)
	labels	a MyLabels
	announcer	an Announcer
	tasks	a MyTasks
	annotations	a MyUserStoryAnnotations
	acceptanceCriteria	sample criteria
	status	Status: Not started

Figure 61: A sample in-IDE live Kanban board

A Kanban board can serve as project documentation that helps product owners track the live progress of the project. In Figure 61, we present a sample Kanban board composed from existing user stories. Various user stories are grouped into three columns: *Not started*, *In progress*, and *Done*, sorted according to their current implementation status. Each user story is a first-class entity and after clicking on the story card, its details are accessible right next to the card representation.

6.1.2 An interactive tutorial

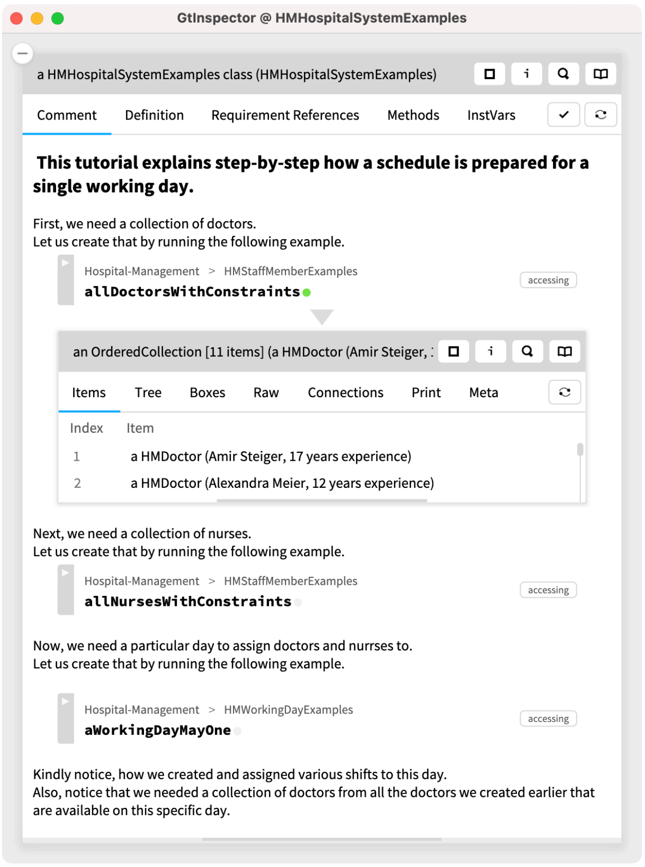
Another example of documentation is tutorials. Tutorials explain various things, *e.g.*, an algorithm, a functionality, even a programming language, step by step to users. Tutorials that involve programming are largely available as video tutorials or blog posts that show a similar pattern: textual documents with static code snippets. There exist online platforms that provide interactive tutorials where users can copy-paste small code snippets in their editor and subsequently explore the execution results. However, such tools and services are limited in the functionality they provide, and cannot be used to explain complicated domain-specific details. In Figure 62, we show an interactive tutorial that explains an algorithm that assigns medical staff to a schedule for one day. This document embeds the already created executable scenarios with supporting text. A user executes scenarios inline and explores the results right next to it without losing the context.

6.2 Agile modeling

Another implication of *Citizen requirements* is on domain modeling. We can use the same building blocks to automate guided assistance through the best practices of the object-oriented design to identify, add, and describe domain entities. We imagine two main steps in such a modeling workflow: (1) analyzing the textual requirements, and (2) describing the behavior of identified domain entities.

6.2.1 Analyze the textual requirements

The object-oriented design process suggests to take the nouns and verbs from the requirement as a starting point for domain modeling, and as candidate classes and responsibilities [145]. Therefore, the first step in our envisioned workflow is to identify domain entities from the textual requirements. In Figure 63, we show how domain experts could annotate a word in any textual requirements document (in this case, a user story) to identify it as a domain concept. However, the initial list of identified entities must periodically be sanitized to form the final list, which contains stable domain abstractions. For instance, consider the following two user stories: *As a waiter, I want to add Cappuccino to an order*, and *As a customer, I want to order pizza to take away.* Although `Cappuccino` and `Pizza` are nouns and potential domain entities, both are kinds of menu items. The modeling workflow must assist domain experts in identifying and recording such relationships among identified domain entities. At the end of the analysis step, we should have a sanitized pool of domain entities to which everyone agrees. What we see in the right-hand side window of Figure 64 is a pool of entities after a user has annotated multiple words from textual requirements. This *view* allows users to specify the *kind-of* relationships between identified entities. Borrowing the terminology from domain-driven design, in this step, the team engages in creating a ubiquitous language [54]. The language (and model) evolves as the team's understanding of the domain grows.


GtInspector @ HMHospitalSystemExamples
a HMHospitalSystemExamples class (HMHospitalSystemExamples)
Comment
Definition
Requirement References
Methods
InstVars

This tutorial explains step-by-step how a schedule is prepared for a single working day.

First, we need a collection of doctors.
Let us create that by running the following example.

Hospital-Management > HMStaffMemberExamples
`allDoctorsWithConstraints`
accessing

an OrderedCollection [11 items] (a HMDoctor (Amir Steiger, :
Items
Tree
Boxes
Raw
Connections
Print
Meta

Index	Item
1	a HMDoctor (Amir Steiger, 17 years experience)
2	a HMDoctor (Alexandra Meier, 12 years experience)

Next, we need a collection of nurses.
Let us create that by running the following example.

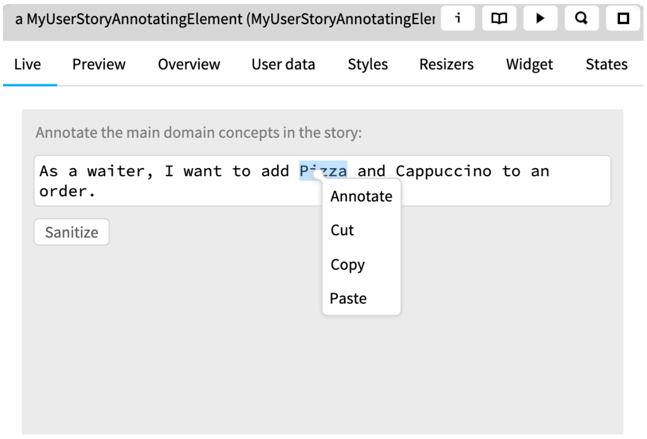
Hospital-Management > HMStaffMemberExamples
`allNursesWithConstraints`
accessing

Now, we need a particular day to assign doctors and nurrses to.
Let us create that by running the following example.

Hospital-Management > HMWorkingDayExamples
`aWorkingDayMayOne`
accessing

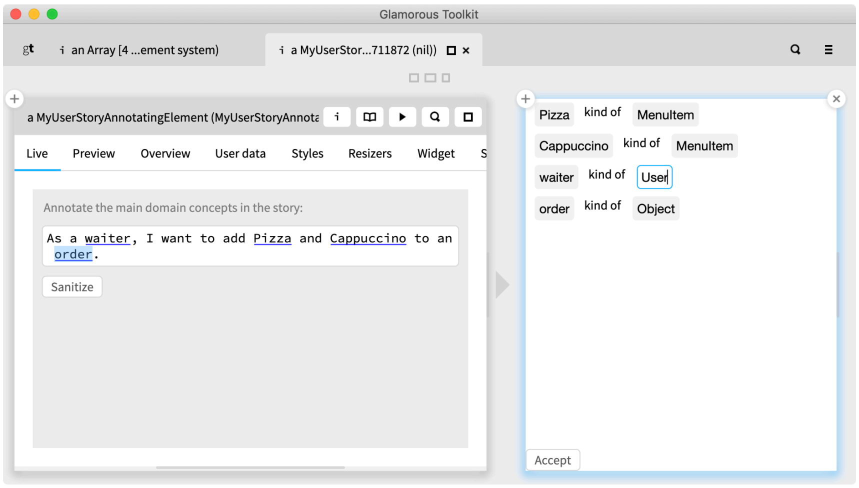
Kindly notice, how we created and assigned various shifts to this day.
Also, notice that we needed a collection of doctors from all the doctors we created earlier that are available on this specific day.

Figure 62: Interactive in-IDE live tutorial


a MyUserStoryAnnotatingElement (MyUserStoryAnnotatingEle
Live
Preview
Overview
User data
Styles
Resizers
Widget
States
Annotate the main domain concepts in the story:
Sanitize
Annotate
Cut
Copy
Paste

```
As a waiter, I want to add Pizza and Cappuccino to an
order.
```

Figure 63: An annotated user story


Glamorous Toolkit
gt
an Array [4 ...ement system)
a MyUserStor...711872 (nil))
a MyUserStoryAnnotatingElement (MyUserStoryAnnota
Live
Preview
Overview
User data
Styles
Resizers
Widget
Annotate the main domain concepts in the story:
Sanitize
Accept

```
As a waiter, I want to add Pizza and Cappuccino to an
order.
```

Pizza	kind of	MenuItem
Cappuccino	kind of	MenuItem
waiter	kind of	User
order	kind of	Object

Figure 64: A pool of sanitized entities and their relationships

6.2.2 Describe the responsibilities of domain entities

In the next step, users describe the responsibilities of the identified entities. The responsibilities of an entity is the information it holds and services it provides to other entities. Users could use a dedicated view that allows them to edit the details (*i.e.*, provided data and services) of the identified entities. In Figure 65, in the rightmost window we see a view that lets users specify data fields for a domain concept.

It is necessary to make sure that an entity is in a valid state after its construction and before and after invoking every public method. Design

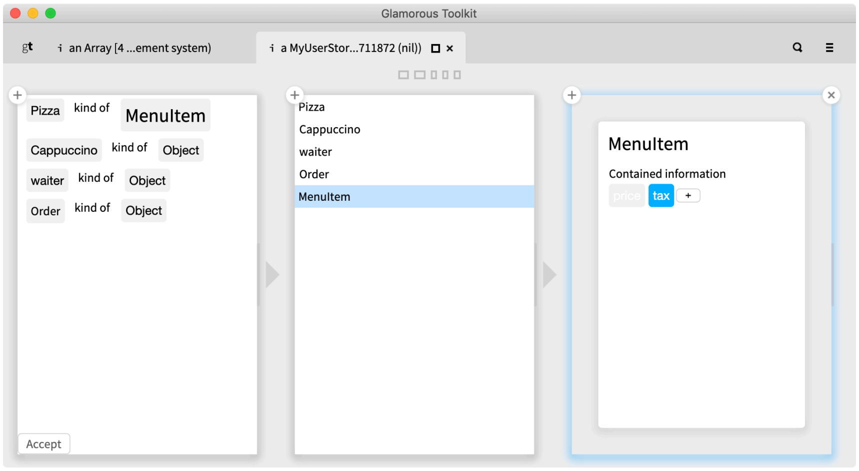


Figure 65: Graphically editing a domain concept

by contract is an approach to define formal, precise and verifiable interface specifications [85]. These specifications are referred to as "contracts." Such explicit contracts bind the client entities to pose valid requests and bind the provider entities to provide the service correctly. With our envisioned agile modeling workflow, we also intend to support domain experts in specifying contracts for the identified entities. With some improvements to the user interface, users can add a textual description to each entity. With such a description, modelers can specify any constraints. It is up to the developers to decide how to guarantee the design by contract in their implementation so that they do not compromise the performance.

Finally, a user must also specify a list of messages an entity understands and reacts to. Pharo is based on the concept of sending messages to objects. This reflects the idea that objects are responsible for their own actions and that the method associated with the message is looked up dynamically. When sending a message to an object, the object, and not the sender, selects the appropriate method for responding to your message. In most cases, the method with the same name as the message is executed [50]. For example, for a "waiter" entity, some of the possible messages could be: `receiveOrderFromTable`, `forwardOrder`, and `deliverOrderToTable`. Modelers can attach a textual description to each identified message as well, which elaborates its content and intention. It is up to the developers to decide how to implement the identified messages as methods.

6.3 Conclusion

In this chapter, we discussed two main implications of the citizen requirements approach. We showed how the discussed building blocks of citizen requirements can help us create and maintain an in-IDE live documentation. We compose documentation out of existing artifacts. Next, we showed how the same building blocks can help us design a modeling workflow that will automate guided assistance through the best practices of the object-oriented design. We believe that compared to existing modeling approaches, such a lightweight modeling workflow is a better alternative to agile domain modeling.

Chapter 7

Conclusions

With more software development companies adopting agile methodologies, it has become crucial to support collaboration among diverse stakeholders to facilitate rapid development and feedback loops. Developing more specialized tools and integrating them into development toolchains is of little help for collaboration. Employing distinct tools requires a tremendous amount of effort and coordination to make those tools interact with each other and establish traceability among numerous artifacts produced in them. Below we elaborate on the contributions that resulted from our research and describe future directions.

7.1 Contributions

Our contributions are the following:

1. We systematically studied the current landscape of requirements engineering tools. We found that at present, a dedicated set of tools support a particular requirements engineering activity and there is a growing tool specialization. Notwithstanding, most of the studied tools were early prototypes that were unavailable to the research community or industry as of now. We present the results of an analysis of 62 software development-related artifacts. We highlight that an artifact is often mutated in different development phases and is used by various stakeholders for distinct reasons. We also analyzed the contents of 1,572 Gherkin files from 23 open-source projects and features of 13 open-source BDD tools. We found that the current support for BDD is not adequate for non-technical stakeholders (chapter 2).

2. We discuss the building blocks that can be implemented in any IDE to adapt it as a software and requirements engineering plat-

form (chapter 3). These building blocks help support stakeholders with agile development.

3. We discuss and demonstrate an approach to create and maintain software-related artifacts as first-class entities in an IDE (chapter 4).
4. We discuss and demonstrate an alternative approach to specify and verify domain entity behavior interactively and visually in an IDE (chapter 5).
5. We discuss the implications of our proposed approach on project documentation and domain modeling (chapter 6).

7.2 Future work

We believe that in requirements and software engineering there are several avenues to follow up. As our next steps, we imagine several improvements to the citizen requirements approach.

7.2.1 Empirical studies

There are various opportunities to conduct empirical studies.

Requirements engineering

There is no comprehensive information available about the current collaborative requirements engineering tools and features they provide. An empirical study investigating the current collaborative RE tools landscape would be a novel contribution to the research community. Likewise, there are various possible extensions of our SLR, the obvious being investigating the underlying purposes and features of the identified tools.

Behavior-driven development

Our analysis of Gherkin feature files makes us curious to understand the practitioners' viewpoint about Gherkin syntax. Likewise, we would also like to investigate the efforts needed to maintain Gherkin feature files together with the test cases.

Agile modeling

Although there exist few empirical studies that discuss collaborative modeling, to the best of our knowledge, no empirical study regarding tool support for agile modeling exists. Such a study will characterize the current advances and support for modeling and contrast and compare it with the traditional approaches.

Documentation

Although there exist few studies on knowledge-based documentation and documentation in agile practices, to the best of our knowledge, a comprehensive overview of live documentation strategies and approaches is still missing. Such a study can help us characterize the potentials and limitations of the proposed approaches and compare ours with the existing ones.

Collaborative tools

Although there exist several tool survey studies, no recent one, to the best of our knowledge, has analyzed contemporary collaborative development tools and how they support various stakeholders in agile development. Such an empirical study will not only allow us to report the specifics of collaborative features but also the limitations and barriers they pose to agile development.

7.2.2 Improving the existing infrastructure

In this doctoral thesis, we demonstrated sample implementation of various research ideas. Going forward, a significant work can be done in various directions.

User interface design

We need to determine through an empirical study the interaction necessities of non-technical stakeholders while using an IDE. The knowledge gathered through such a study will help us design usable GUIs for artifact creation, exploration, and manipulation. We must also put effort into designing GUIs that could be useful across more number application domains.

Extending the current support

We can design a more stakeholder-inclusive workflow by implementing a wide variety of artifacts. Likewise, the existing behavior verification and modeling workflow can also be improved by adding intermediate steps and corresponding GUIs.

Extending DevOps cycle

As we maintain various artifacts— produced in the early development phases— together with the source code, the typical DevOps cycle can be adapted further to include additional phases as shown in Figure 71.

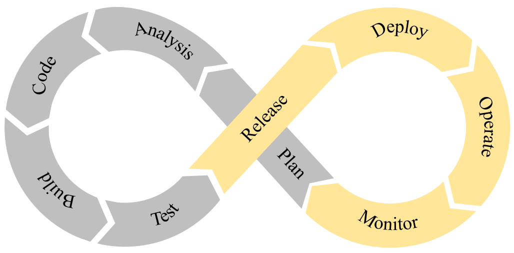


Figure 71: A modified DevOps cycle

7.2.3 Evaluation in a practical setting

Although a pilot survey gave us confidence that our proposed approach is intriguing for practitioners, we need to conduct other types of evaluations. Particularly, we need a user study to analyze how practitioners use such a workflow in a real setting and compare time and effort needed to accomplish various tasks in traditional and proposed settings. We also need to conduct a user study to compare the efforts and time needed to do BDD in traditional and proposed settings.

7.3 Summary and conclusion

By examining the existing literature, we found that existing tool support for engineering requirements and behavior verification focuses on individual RE activities and lacks collaborative features. In this doctoral thesis, we argued that we should refrain from using independent tools and proposing toolchains to support stakeholders with agile development. With this approach, numerous software-related artifacts scatter among employed tools, and establishing traceability between artifacts becomes cumbersome. Instead, we propose to adapt an IDE to be a platform for both requirements and software engineering. With our approach, we connect requirements at various levels of abstraction to each other and to the application source code through object-oriented relationships. We discuss the building blocks that can be implemented in any reflective IDE with a rich-enough graphical engine. To enable such an approach in a software world, we demonstrate how we can provide in-IDE support for various stakeholders for creating and manipulating various software-related artifacts. Similarly, we compose live executable documentation from the existing artifacts. We demonstrate how to provide support for non-technical stakeholders for creating and manipulating business entities directly in an IDE through appropriate graphical interfaces. Finally, we demonstrate how to provide support for modeling rapidly evolving domains and verifying the application behavior.

We believe that with this work, in the future, we will approach the software development process with a fresh mind. We will motivate researchers to look at traceability and documentation problems from a different perspective. Likewise, we will motivate researchers to look at collaboration from a process perspective rather than a tool perspective.

Bibliography

[1] Enterprise architect. https://sparxsystems.com/. Accessed: 2021-04-03.

[2] Gherkin reference. https://github.com/cucumber/common/blob/main/gherkin/CHANGELOG.md. Accessed: 2021-04-03.

[3] Glamorous toolkit. http://gtoolkit.com/. Accessed: 2021-04-03.

[4] Lucid chart. https://www.lucidchart.com/pages/examples/uml_diagram_tool. Accessed: 2021-04-03.

[5] Rational doors. https://www.ibm.com/products/requirements-management. Accessed: 2021-04-03.

[6] Ulrike Abelein and Barbara Paech. A proposal for enhancing user-developer communication in large it projects. In *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)*, pages 1–3. IEEE, 2012.

[7] Philip Achimugu, Ali Selamat, Roliana Ibrahim, and Mohd Naz'ri Mahrin. A systematic literature review of software requirements prioritization research. *Information and software technology*, 56(6):568–585, 2014.

[8] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1199–1210. IEEE, 2019.

[9] Diego Albuquerque, Bruno Cafeo, Alessandro Garcia, Simone Barbosa, Silvia Abrahao, and António Ribeiro. Quantifying usability of domain-specific languages: An empirical study on software maintenance. *Journal of Systems and Software*, 101:245–259, 2015.

[10] Nasir Ali, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. *IEEE Transactions on Software Engineering*, 39(5):725–741, 2012.

[11] Vander Alves, Nan Niu, Carina Alves, and George Valença. Requirements engineering for software product lines: A systematic literature review. *Information and Software Technology*, 52(8):806–820, 2010.

[12] Scott Ambler. *Agile modeling: effective practices for extreme programming and the unified process*. John Wiley & Sons, 2002.

[13] Talat Ambreen, Naveed Ikram, Muhammad Usman, and Mahmood Niazi. Empirical research in requirements engineering: trends and opportunities. *Requirements Engineering*, 23(1):63–95, 2018.

[14] Pablo Oliveira Antonino, Thorsten Keuler, Nicolas Germann, and Brian Cronauer. A non-invasive approach to trace architecture design, requirements specification and agile artifacts. In *2014 23rd Australian Software Engineering Conference*, pages 220–229. IEEE, 2014.

[15] Hazeline U Asuncion and Richard N Taylor. Automated techniques for capturing custom traceability links across heterogeneous artifacts. In *Software and Systems Traceability*, pages 129–146. Springer, 2012.

[16] Julian M. Bass. Artefacts and agile method tailoring in large-scale offshore software development programmes. *Information and Software Technology*, 75:1 – 16, 2016.

[17] Marina Berkovich, Sebastian Esch, Christian Mauro, Jan Marco Leimeister, and Helmut Krcmar. Towards an artifact model for requirements to it-enabled product service systems. 2011.

[18] Paul Beynon-Davies, Chris Carne, Hugh Mackay, and Douglas Tudhope. Rapid application development (rad): an empirical review. *European Journal of Information Systems*, 8(3):211–223, 1999.

[19] Niels Bik, Garm Lucassen, and Sjaak Brinkkemper. A reference method for user story requirements in agile systems development. In *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, pages 292–298. IEEE, 2017.

[20] Leonard Peter Binamungu, Suzanne M Embury, and Nikolaos Konstantinou. Detecting duplicate examples in behaviour driven development specifications. In *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 6–10. IEEE, 2018.

[21] Leonard Peter Binamungu, Suzanne M Embury, and Nikolaos Konstantinou. Maintaining behaviour driven development specifications: Challenges and opportunities. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 175–184. IEEE, 2018.

[22] Leonard Peter Binamungu, Suzanne M Embury, and Nikolaos Konstantinou. Characterising the quality of behaviour driven development specifications. In *International Conference on Agile Software Development*, pages 87–102. Springer, Cham, 2020.

[23] Steve Boßelmann, Markus Frohme, Dawid Kopetzki, Michael Lybecait, Stefan Naujokat, Johannes Neubauer, Dominic Wirkner, Philip Zweihoff, and Bernhard Steffen. Dime: a programming-less modeling environment for web applications. In *International Symposium on Leveraging Applications of Formal Methods*, pages 809–832. Springer, 2016.

[24] Juan Boubeta-Puig, Guadalupe Ortiz, and Inmaculada Medina-Bulo. Model4cep: Graphical domain-specific modeling languages for cep domains and event patterns. *Expert Systems with Applications*, 42(21):8095–8110, 2015.

[25] Janis A Bubenko. Challenges in requirements engineering. In *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*, pages 160–162. IEEE, 1995.

[26] Thomas Buchmann. Towards tool support for agile modeling: sketching equals modeling. In *Proceedings of the 2012 Extreme Modeling Workshop*, pages 9–14, 2012.

[27] Thomas Buchmann and Sandra Greiner. Handcrafting a triple graph transformation system to realize round-trip engineering between uml class models and java source code. In *International Conference on Software Paradigm Trends*, volume 2, pages 27–38. SCITEPRESS, 2016.

[28] Margaret M Burnett, John Wesley Atwood, and Zachary T Welch. Implementing level 4 liveness in declarative visual programming languages. In *Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No. 98TB100254)*, pages 126–133. IEEE, 1998.

[29] Manuel F Caro, Darsana P Josyula, Jovani A Jiménez, Catriona M Kennedy, and Michael T Cox. A domain-specific visual language for modeling metacognition in intelligent systems. *Biologically Inspired Cognitive Architectures*, 13:75–90, 2015.

[30] Laura Carvajal, Ana M Moreno, Maria-Isabel Sanchez-Segura, and Ahmed Seffah. Usability through software design. *IEEE Transactions on Software Engineering*, 39(11):1582–1596, 2013.

[31] Glenn Cavarlé, Alain Plantec, Steven Costiou, and Vincent Ribaud. Dynamic round-trip engineering in the context of fomdd. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, pages 1–7, 2016.

[32] Adwait Chandorkar, Nitish Patkar, Andrea Di Sorbo, and Oscar Nierstrasz. An exploratory study on the usage of gherkin features in open-source projects. 2022.

[33] Betty HC Cheng and Joanne M Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, 2007.

[34] Sridhar Chimalakonda and Dan Hyung Lee. On the evolution of software and systems product line standards. *ACM SIGSOFT Software Engineering Notes*, 41(3):27–30, 2016.

[35] Andrei Chis. *Moldable tools*. Lulu. com, 2016.

[36] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. The moldable inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 44–60, 2015.

[37] Rance Cleaveland. Programming is modeling. In *International Symposium on Leveraging Applications of Formal Methods*, pages 150–161. Springer, 2018.

[38] Oliver Creighton, Martin Ott, and Bernd Bruegge. Software cinema-video-based requirements engineering. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 109–118. IEEE, 2006.

[39] Cucumber. Tool website at `https://cucumber.io/`. Accessed: 2020-11-22.

[40] Robertas Damaševičius. Analysis of software design artifacts for socio-technical aspects. *INFOCOMP Journal of Computer Science*, 6(4):7–16, 2007.

[41] Joern David, Maximilian Koegel, Helmut Naughton, and Jonas Helming. Traceability rearmed. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 340–348. IEEE, 2009.

[42] Alan Davis, Oscar Dieste, Ann Hickey, Natalia Juristo, and Ana M Moreno. Effectiveness of requirements elicitation techniques: Empirical results derived from a systematic review. In *14th IEEE International Requirements Engineering Conference (RE'06)*, pages 179–188. IEEE, 2006.

[43] Alan Davis, Ann Hickey, Oscar Dieste, Natalia Juristo, and Ana Moreno. A quantitative assessment of requirements engineering publications–1963–2006. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 129–143. Springer, 2007.

[44] Juan M Carrillo de Gea, Joaquín Nicolás, José L Fernández Alemán, Ambrosio Toval, Christof Ebert, and Aurora Vizcaíno. Requirements engineering tools. *IEEE software*, 28(4):86–91, 2011.

[45] Juan M Carrillo De Gea, Joaquín Nicolás, José L Fernández Alemán, Ambrosio Toval, Christof Ebert, and Aurora Vizcaíno. Requirements engineering tools: Capabilities, survey and assessment. *Information and Software Technology*, 54(10):1142–1157, 2012.

[46] Robert DeLine and Danyel Fisher. Supporting exploratory data analysis with live programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 111–119. IEEE, 2015.

[47] Philipp Diebold and Marc Dahlem. Agile practices in practice: a mapping study. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2014.

[48] Andreas Drechsler. Designing to inform: Toward conceptualizing practitioner audiences for socio-technical artifacts in design science research in the information systems discipline. *Informing Sci. Int. J. an Emerg. Transdiscipl.*, 18:31–47, 2015.

[49] Andreas Drechsler, Alan R Hevner, and T Grandon Gill. Beyond rigor and relevance: Exploring artifact resonance. In *2016 49th Hawaii international conference on system sciences (HICSS)*, pages 4434–4443. IEEE, 2016.

[50] Ducasse. Pharo by example `https://books.pharo.org/updated-pharo-by-example/`. Accessed: 2021-30-11.

[51] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[52] Peter Eeles and Oliver Sims. *Building business objects.* Wiley Publishing, 1998.

[53] David W Embley and Bernhard Thalheim. *Handbook of conceptual modeling: theory, practice, and research challenges.* Springer, 2012.

[54] Eric Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[55] Markus Gaelli, Rafael Wampfler, and Oscar Nierstrasz. Composing tests from examples. *J. Object Technol.*, 6(9):71–86, 2007.

[56] Uri Gal, Kalle Lyytinen, and Youngjin Yoo. The dynamics of it boundary objects, information infrastructures, and organisational identities: the introduction of 3d modelling technologies into the architecture, engineering, and construction industry. *European journal of information systems*, 17(3):290–304, 2008.

[57] Andrei Garcia, Tiago Silva da Silva, and Milene Selbach Silveira. Artifacts for agile user-centered design: a systematic mapping. *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.

[58] Martin Glinz. Should requirements be objects? In *Tutorial Position Paper, 14th Annual International Symposium on Systems Engineering.* Citeseer, 2004.

[59] Fahad R Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Using free modeling as an agile method for developing domain specific modeling languages. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 24–34, 2016.

[60] Orlena Gotel and Anthony Finkelstein. Contribution structures [requirements artifacts]. In *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*, pages 100–107. IEEE, 1995.

[61] Orlena CZ Gotel and CW Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering*, pages 94–101. IEEE, 1994.

[62] Orlena CZ Gotel, Francis T Marchese, and Stephen J Morris. On requirements visualization. In *Second International Workshop on Requirements Engineering Visualization (REV 2007)*, pages 11–11. IEEE, 2007.

[63] Lea Hänsenberger. Jexample-extending junit with explicit dependencies. 2008.

[64] Rashina Hoda and James Noble. Becoming agile: a grounded theory of agile transitions in practice. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 141–151. IEEE, 2017.

[65] Manuel Imaz and David Benyon. How stories capture interactions. In *INTERACT*, volume 99, pages 321–328, 1999.

[66] Irum Inayat, Siti Salwah Salim, Sabrina Marczak, Maya Daneva, and Shahaboddin Shamshirband. A systematic literature review on agile requirements engineering practices and challenges. *Computers in human behavior*, 51:915–929, 2015.

[67] Marcin Jamro and Dariusz Rzonca. Agile and hierarchical round-trip engineering of iec 61131-3 control software. *Computers in Industry*, 96:1–9, 2018.

[68] JDave. Tool repository at `https://github.com/jdave/JDave`. Accessed: 2020-06-19.

[69] JIRA. Tool website at `https://www.atlassian.com/software/jira`. Accessed: 2020-11-22.

[70] Hermann Kaindl. The missing link in requirements engineering. *ACM SIGSOFT Software Engineering Notes*, 18(2):30–39, 1993.

[71] Hyeonsu Kang and Philip J Guo. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 737–745, 2017.

[72] Javed Ali Khan, Lin Liu, Lijie Wen, and Raian Ali. Crowd intelligence in requirements engineering: Current status and future directions. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 245–261. Springer, 2019.

[73] Barbara Kitchenham, O Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology*, 51(1):7–15, 2009.

[74] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

[75] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[76] Adrian Kuhn, Bart Van Rompaey, Lea Haensenberger, Oscar Nierstrasz, Serge Demeyer, Markus Gaelli, and Koenraad Van Leemput. Jexample: Exploiting dependencies between tests to improve defect localization. In *International Conference on Agile Processes and Extreme Programming in Software Engineering*, pages 73–82. Springer, 2008.

[77] Remo Lemma and Michele Lanza. Co-evolution as the key for live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 9–10. IEEE, 2013.

[78] Rakesh Kumar Lenka, Srikant Kumar, and Sunakshi Mamgain. Behavior driven development: Tools and challenges. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pages 1032–1037. IEEE, 2018.

[79] Timothy C Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE software*, 20(6):35–39, 2003.

[80] LightBDD. Tool repository at `https://github.com/LightBDD/LightBDD`. Accessed: 2020-06-19.

[81] Olga Liskin. How artifacts support and impede requirements communication. In Samuel A. Fricker and Kurt Schneider, editors, *Requirements Engineering: Foundation for Software Quality*, pages 132–147, Cham, 2015. Springer International Publishing.

[82] Garm Lucassen, Fabiano Dalpiaz, Jan Martijn EM van der Werf, and Sjaak Brinkkemper. The use and effectiveness of user stories in practice. In *International working conference on requirements engineering: Foundation for software quality*, pages 205–222. Springer, 2016.

[83] Imran Mahmud and Vito Veneziano. Mind-mapping: An effective technique to facilitate requirements engineering in agile software development. In *14th International Conference on Computer and Information Technology (ICCIT 2011)*, pages 157–162. IEEE, 2011.

[84] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. A systematic literature review of software visualization evaluation. *Journal of Systems and Software*, 144:165–180, 2018.

[85] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[86] David Mosteller, Lawrence Cabac, and Michael Haustermann. Integrating petri net semantics in a model-driven approach: The renew meta-modeling and transformation framework. In *Transactions*

on Petri Nets and Other Models of Concurrency XI, pages 92–113. Springer, 2016.

[87] MSpec. Tool repository at `https://github.com/machine/machine.specifications`. Accessed: 2020-06-19.

[88] Brendan Murphy, Christian Bird, Thomas Zimmermann, Laurie Williams, Nachiappan Nagappan, and Andrew Begel. Have agile techniques been the silver bullet for software development at microsoft? In *2013 ACM/IEEE international symposium on empirical software engineering and measurement*, pages 75–84. IEEE, 2013.

[89] Leckraj Nagowah, Zarah Goolfee, and Chris Bergue. Rtet-a round trip engineering tool. In *2013 International Conference of Information and Communication Technology (ICoICT)*, pages 381–387. IEEE, 2013.

[90] Stefan Naujokat, Michael Lybecait, Dawid Kopetzki, and Bernhard Steffen. Cinco: a simplicity-driven approach to full generation of domain-specific graphical modeling tools. *International Journal on Software Tools for Technology Transfer*, 20(3):327–354, 2018.

[91] NBehave. Tool repository at `https://github.com/nbehave/NBehave`. Accessed: 2020-06-19.

[92] Oscar Nierstrasz. The death of object-oriented programming. In *International Conference on Fundamental Approaches to Software Engineering*, pages 3–10. Springer, 2016.

[93] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM, 2000.

[94] Anton Okolnychyi and Konrad Fögen. A study of tools for behavior-driven development. *Full-scale Software Engineering/Current Trends in Release Engineering*, page 7, 2016.

[95] Carla Pacheco and Ivan Garcia. A systematic literature review of stakeholder identification methods in requirements elicitation. *Journal of Systems and Software*, 85(9):2171–2181, 2012.

[96] Nitish Patkar. Moldable requirements. In *Benevol 2020: Proceedings of the 19th Belgium-Netherlands software evolution workshop*, 2020.

[97] Nitish Patkar, Andrei Chis, Nataliia Stulova, and Oscar Nierstrasz. Interactive behavior-driven development: a low-code perspective. pages 128–137, 2021.

[98] Nitish Patkar, Andrei Chis, Nataliia Stulova, and Oscar Nierstrasz. First-class artifacts as building blocks for live in-ide documentation. 2022.

[99] Jeff Patton and Peter Economy. *User story mapping: discover the whole story, build the right product.* " O'Reilly Media, Inc.", 2014.

[100] Pawson. Framework website at `http://www.nakedobjects.org/`. Accessed: 2021-12-17.

[101] Richard Pawson and Robert Matthews. Naked objects: a technique for designing more expressive systems. *ACM SIGPLAN Notices*, 36(12):61–67, 2001.

[102] Richard Pawson and Vincent Wade. Agile development using naked objects. In *International Conference on Extreme Programming and Agile Processes in Software Engineering*, pages 97–103. Springer, 2003.

[103] Vito Perrone, Davide Bolchini, and Paolo Paolini. A stakeholders centered approach for conceptual modeling of communication-intensive applications. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 25–33, 2005.

[104] Marian Petre. Uml in practice. In *2013 35th international conference on software engineering (icse)*, pages 722–731. IEEE, 2013.

[105] phpspec (`http://www.phpspec.net/en/stable/`). Tool repository at `https://github.com/phpspec/phpspec`. Accessed: 2020-06-19.

[106] Awais Rashid, Peter Sawyer, Ana Moreira, and João Araújo. Early aspects: A model for aspect-oriented requirements engineering. In *Proceedings IEEE Joint International Conference on Requirements Engineering*, pages 199–202. IEEE, 2002.

[107] Sukanya Ratanotayanon, Susan Elliott Sim, and Derek J Raycraft. Cross-artifact traceability using lightweight links. In *2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 57–64. IEEE, 2009.

[108] Mohammad S Raunak and David Binkley. Agile and other trends in software engineering. In *2017 IEEE 28th Annual Software Technology Conference (STC)*, pages 1–7. IEEE, 2017.

[109] Lukas Renggli, Stéphane Ducasse, and Adrian Kuhn. Magritte–a meta-driven approach to empower developers and end users. In *International Conference on Model Driven Engineering Languages and Systems*, pages 106–120. Springer, 2007.

[110] Clay Richardson and John R Rymer. New development platforms emerge for customer-facing applications. 2014.

[111] Clay Richardson and John R Rymer. Vendor landscape: The fractured, fertile terrain of low-code application platforms. *FORRESTER, Janeiro*, 2016.

[112] Apurvanand Sahay, Arsene Indamutsa, Davide Di Ruscio, and Alfonso Pierantonio. Supporting the understanding and comparison of low-code development platforms. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 171–178. IEEE, 2020.

[113] E.-M. Schön, J. Thomaschewski, and M. J. Escalona. Identifying agile requirements engineering patterns in industry. Association for Computing Machinery, 2017.

[114] Eva-Maria Schön, Jörg Thomaschewski, and María José Escalona. Agile requirements engineering: A systematic literature review. *Computer Standards & Interfaces*, 49:79–91, 2017.

[115] Shane Sendall and Jochen Küster. Taming model round-trip engineering. In *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, volume 1. Citeseer, 2004.

[116] Björn Senft, Holger Fischer, Simon Oberthür, and Nitish Patkar. Assist users to straightaway suggest and describe experienced problems. In *International Conference of Design, User Experience, and Usability*, pages 758–770. Springer, 2018.

[117] Atif Shah, Mohamed Ali Alasow, Faisal Sajjad, and Jawad Javed Akbar Baig. An evaluation of software requirements tools. In *2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*, pages 278–283. IEEE, 2017.

[118] Thiago Rocha Silva, Jean-Luc Hak, and Marco Winckler. Testing prototypes and final user interfaces through an ontological perspective for behavior-driven development. In *Human-Centered and Error-Resilient Systems Development*, pages 86–107. Springer, 2016.

[119] Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 269–287. Springer, 2012.

[120] Carlos Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387. IEEE, 2011.

[121] Ian Sommerville. Software documentation. *Software engineering*, 2:143–154, 2001.

[122] Specs2. Tool repository at `https://etorreborre.github.io/specs2/`. Accessed: 2020-06-19.

[123] Christoph Johann Stettina, Werner Heijstek, and Tor Erlend Fægri. Documentation work in agile teams: The role of documentation formalism in achieving a sustainable practice. In *2012 Agile Conference*, pages 31–40. IEEE, 2012.

[124] Christoph Johann Stettina and Egbert Kroon. Is there an agile handover? an empirical study of documentation and project handover practices across agile software teams. In *2013 International Conference on Engineering, Technology and Innovation (ICE) & IEEE International Technology Management Conference*, pages 1–12. IEEE, 2013.

[125] Alistair G Sutcliffe, Neil AM Maiden, Shailey Minocha, and Darrel Manuel. Supporting scenario-based requirements engineering. *IEEE Transactions on software engineering*, 24(12):1072–1088, 1998.

[126] Ivan E Sutherland. Sketchpad a man-machine graphical communication system. *Simulation*, 2(5):R–3, 1964.

[127] Steven L Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

[128] Michael Alexander Tröls, Atif Mashkoor, and Alexander Egyed. Multifaceted consistency checking of collaborative engineering artifacts. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 278–287. IEEE, 2019.

[129] Bernhard Turban. *Tool-Based Requirement Traceability Between Requirement and Design Artifacts*. Springer Science & Business Media, 2013.

[130] Easyb (`http://easyb.io/v1/index.html`). Accessed: 2020-06-19.

[131] JGiven (`http://jgiven.org`). Tool repository at `https://github.com/TNG/JGiven`. Accessed: 2020-06-19.

[132] RSpec (`http://rspec.info`). Tool repository at `https://github.com/rspec`. Accessed: 2020-06-19.

[133] StoryQ (`https://archive.codeplex.com/?p=storyq`). Accessed: 2020-06-19.

[134] Concordion (https://concordion.org). Accessed: 2020-06-19.

[135] Gauge (https://gauge.org). Tool repository at https://github.com/getgauge/gauge. Accessed: 2020-06-19.

[136] JBehave (https://jbehave.org). Tool repository at https://github.com/jbehave/jbehave-core. Accessed: 2020-06-19.

[137] Spock (http://spockframework.org/). Tool repository at https://github.com/spockframework/spock. Accessed: 2020-06-19.

[138] SpecFlow (https://specflow.org). Tool repository at https://github.com/SpecFlowOSS/SpecFlow. Accessed: 2020-06-19.

[139] BDDfy (https://teststackbddfy.readthedocs.io/en/latest/). Accessed: 2020-06-19.

[140] ScalaTest (http://www.scalatest.org/). Tool repository at https://github.com/scalatest/scalatest. Accessed: 2020-06-19.

[141] Ken Vanherpen, Joachim Denil, Hans Vangheluwe, and Paul De Meulenaere. Model transformations for round-trip engineering in control deployment co-design. In *SpringSim (TMS-DEVS)*, pages 55–62, 2015.

[142] Niksa Visic, Hans-Georg Fill, Robert Andrei Buchmann, and Dimitris Karagiannis. A domain-specific language for modeling method definition: From requirements to grammar. In *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*, pages 286–297. IEEE, 2015.

[143] Jiawei Wang, Li Li, and Andreas Zeller. Better code, better sharing: on the need of analyzing jupyter notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, pages 53–56, 2020.

[144] Dietmar Winkler, Richard Mordinyi, and Stefan Biffl. Research prototypes versus products: lessons learned from software development processes in research projects. In *European Conference on Software Process Improvement*, pages 48–59. Springer, 2013.

[145] Rebecca Wirfs-Brock and Brian Wilkerson. Object-oriented design: A responsibility-driven approach. *ACM sigplan notices*, 24(10):71–75, 1989.

[146] Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers.* Pragmatic Bookshelf, 2017.

[147] Aidan ZH Yang, Daniel Alencar da Costa, and Ying Zou. Predicting co-changes between functionality specifications and source code in behavior driven development. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 534–544. IEEE, 2019.

[148] Anna Zaitsev, Uri Gal, and Barney Tan. Coordination artifacts in agile software development. *Information and Organization*, 30(2):100288, 2020.

[149] Fiorella Zampetti, Andrea Di Sorbo, Corrado Aaron Visaggio, Gerardo Canfora, and Massimiliano Di Penta. Demystifying the adoption of behavior-driven development in open source projects. *Information and Software Technology*, page 106311, 2020.

[150] Xiong Zhang and Philip J Guo. Ds. js: Turn any webpage into an example-centric live programming environment for learning data science. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 691–702, 2017.

[151] Philip Zweihoff, Stefan Naujokat, and Bernhard Steffen. Pyro: Generating domain-specific collaborative online modeling environments. In *International Conference on Fundamental Approaches to Software Engineering*, pages 101–115. Springer, 2019.

Appendix A

Appendix

A.1 SLR methodology

In Figure A1, we sketch the process we followed. We assigned each of our activities to one of the three SLR-related phases: *planning*, *conducting the review*, and *reporting*.

A.1.1 Planning phase

In the planning phase, we established the objectives of the SLR, *i.e.*, (i) to identify publications that report a tool supporting one of the several RE activities, *i.e.*, elicitation, analysis, documentation, validation/negotiation, and management, according to the CPRE syllabus, and (ii) to study the identified tools from multiple dimensions as mentioned in the next subsection.

Specification of Research Questions

In conclusion, we specified the following two research questions:

- **RQ$_1$**: *What are the characteristics of RE tools?*
  We report our findings across three dimensions: (i) the publication trends (*e.g.*, publication venues, citations), (ii) the scope (*e.g.*, supported RE activities, types of the tools), and (iii) the tool availability (*e.g.*, webpage and source code repository URLs).

- **RQ$_2$**: *How mature are RE tools?*
  We present a "goal, question, metric" (GQM) approach to evaluate the rigor and relevance of the evaluations to the field of RE. Additionally, we report on the rigor and relevance scores of the included publications.

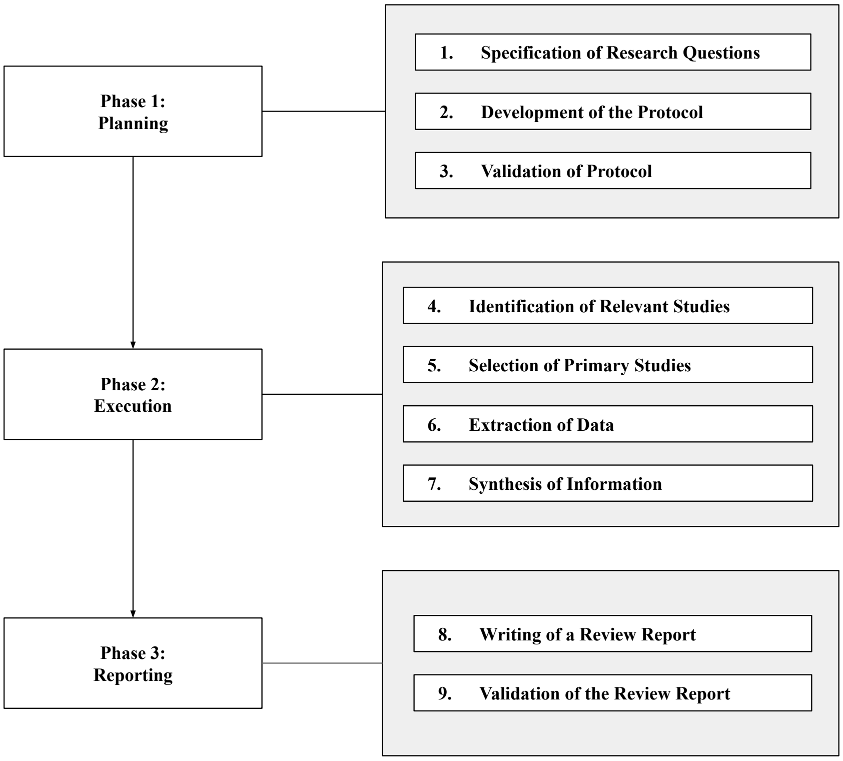

Phase 1:
Planning
Phase 2:
Execution
Phase 3:
Reporting
1. Specification of Research Questions
2. Development of the Protocol
3. Validation of Protocol
4. Identification of Relevant Studies
5. Selection of Primary Studies
6. Extraction of Data
7. Synthesis of Information
8. Writing of a Review Report
9. Validation of the Review Report

Figure A1: SLR process

Development of the survey protocol

The survey protocol defines the activities required to carry out an SLR. It helps to reduce researcher bias, and it defines the process of selecting the data sources, searching them, and synthesizing the obtained information.

Data sources. SLRs often select as their data source digital libraries, such as ACM DL[1] or IEEE Xplore.[2] To find relevant primary studies for analysis, they define a search strategy that typically is based on keywords related to the topic of interest. Instead, we decided to adopt as our data source complete set of papers published by the top SE and RE conferences. We believe that hundreds of papers dedicated to requirements engineering, and additionally, to software engineering, constitute a sound body of literature — a strategy employed in similar SLRs [84]. Specifically, we reviewed proceedings from the top nine SE conferences, according to the *Computing Research and Education Association of Australasia* (CORE)

[1]`http://dl.acm.org/`
[2]`http://ieeexplore.ieee.org.`

ranking during the years 2015-2019.[3] We selected the CORE ranking because the executive committee of CORE assigns ranks (*i.e.*, A*, A, B, *etc.*) to conferences in the computing disciplines, and it is well-regarded in the area of SE. We selected the years 2015-2019 as the previous studies were either too old (*i.e.*, from 2011-12) or did not cover important SE venues [44, 45, 117]. We did not include the year 2020 because most of the proceedings for the year 2020 were unavailable when we started the data gathering process. Additionally, we examined all issues from of top SE journals, as summarized in Table A1, released during the same time period.

In Table A1, the column "Type" specifies whether a publication originated from a conference (marked with "C") or a journal ("J"). The column "Rank" indicates the corresponding CORE rank.

Study selection procedure. We systematically selected relevant publications by following two steps: (i) examining the publication title, abstract, keywords, and introduction to find a clear mention of a developed tool, a technique, an approach, or a method that can support one or more of the RE activities, and (ii) filtering the selected publications according to the inclusion and exclusion criteria. We validated the selected publications to mitigate any selection bias. We selected a total of 203 publications.

Inclusion and exclusion criteria. To include a publication, it must: (i) mention an RE tool that clearly originates from the authors, (ii) indicate in the title, keywords, or introduction that the tool can support RE (for the publications from non-RE venues). We excluded publications that: (i) were mapping or SLR studies, (ii) proposed or discussed the same tool from another publication, or (iii) lacked concrete details of a tool. We included journal extensions of original conference papers, as often they provide more details about the evaluation.

Validation of the protocol

The protocol was validated by the study team. In particular, all the members of the team discussed and agreed upon: (i) which data sources must be considered, (ii) which publication selection criteria must be considered, and (iii) the final inclusion and exclusion criteria for the publications.

A.1.2 Execution phase

We followed the following four steps to conduct the actual SLR:

[3] *CORE rankings portal*, accessed March 18, 2020, `http://www.core.edu.au/`

Table A1: Used data sources

Abbr.	Abbreviation in full	Type	Rank
ASE	Automated Software Engineering Conference	C	A
ESEC	European Software Engineering Conference	C	A*
ESEM	International Symposium on Empirical Software Engineering and Measurement	C	A
ICSE	International Conference on Software Engineering	C	A*
ICSME	International Conference on Software Maintenance and Evolution	C	A
MSR	International Working Conference on Mining Software Repositories	C	A
RE	Requirements Engineering International Conference	C	A
REFSQ	International Workshop on Requirements Engineering: Foundations for Software Quality	C	B
SANER	International Conference on Software Analysis, Evolution and Re-engineering	C	A
EMSE	Empirical Software Engineering	J	A
IST	Information and Software Technology	J	A
JSS	Journal of Systems and Software	J	A
RE J	Requirements Engineering Journal	J	B
SoSyM	Software and System Modeling	J	B
TOSEM	Transactions on Software Engineering and Methodology	J	A*
TSE	Transactions on Software Engineering	J	A*

Identification of relevant studies

We searched relevant publications in the research, industry, and tool demo tracks of the included conferences as they are peer reviewed or evaluated by the research community. From 203 publications that we initially selected, 112 (55%) were relevant and included in this work. The publication counts are listed by year in Table A2. The complete list of the selected publications can be found in section A.9.

Selection of primary studies

Publications included in this SLR propose a tool to support at least one of several RE activities. Often, the authors claimed to propose a tool to

Table A2: The distribution of the included studies by venues and years

Venue	2015	2016	2017	2018	2019	Total
ASE	0	2	2	0	2	6
ESEC	1	0	1	0	0	2
ESEM	-	-	-	-	-	-
ICSE	-	-	2	2	7	11
ICSME	-	-	-	-	-	-
MSR	-	-	-	-	-	-
RE	8	7	9	10	5	39
REFSQ	2	2	3	2	1	10
SANER	-	-	-	1	-	1
EMSE	1	-	-	-	1	2
IST	2	4	1	2	4	13
JSS	-	-	1	4	-	5
RE J	2	3	5	7	4	21
SoSyM	-	-	-	1	1	2
TOSEM	-	-	-	-	-	-
TSE	-	-	-	-	-	-

support RE. However, in many other publications — especially from non-RE venues — no such explicit claims are made by authors. Only after thoroughly reading such a publication, could we decide to include it.

Extraction of data

We thoroughly read the included publications to record information about the 32 parameters described in Table A3. These 32 parameters help us to answer RQ_1 and RQ_2, and were selected by the study team through brainstorming and discussion. In case we could not determine a value for the parameters 9 to 19 for a publication, we contacted the publication authors to obtain additional information. The column "D" indicates one of the four dimensions mentioned earlier, *i.e.*, the publication trends, the scope, the availability, and the reliability of performed evaluation. The column "P" denotes the number of the parameter, followed by a description and possible values. The responses from authors are recorded as free text, and are indicated either as a string or a number. After reading a specific publication, if the supported RE activity was not clear, then we read the publication independently and proposed the activity to be assigned. In case we disagreed, we then concluded through brainstorming and discussion the activity to be assigned.

Synthesis of information

We codified the textual responses obtained from the publication authors.

Table A3: Collected data from existing studies

D	P	Description	Possible values
1	1	name of the tool	*string*
	2	publication year	*string*
	3	venue	*as mentioned in Table A1*
	4	track	Research, Industry, Tool demo, Journal
	5	number of citations	*number*
2	6	supported RE activity	elicitation, analysis, specification, validation, management
	7	whether the authors claim it to be a prototype	yes, no
	8	whether the authors claim it to be collaborative	yes, no
	9	the intended audience of the tool	*string*
	10	type of the tool	web, desktop, mobile, plug-in
	11	the list of artifacts the tool works with	*string*
	12	the import and export formats of the artifacts	*string*
	13	the required operating system in case of a desktop or mobile app	*string*
	14	programming languages used to develop the tool	*string*
	15	other technologies or frameworks used to develop the tool	*string*
3	16	URL to access the project or download, installation page	*string*
	17	whether the tool is open-source	yes, no
	18	URL of the source code repository	*string*
	19	last activity on the source code repository	*date*
4	20	whether the authors carried out an evaluation of their tool	yes, no
	21	whether the authors mention any evaluation guidelines	yes, no
	22	the type of evaluation	experiment, case study, survey
	23	number of case studies	*number*
	24	the purpose of evaluation	*string*
	25	evaluation variables	*string*
	26	participant population	*number*
	27	number of participants of an experiment or a survey	*number*
	28	experience of participants in years	*number*
	29	motivation of the participant	*string*
	30	data collection method	first, second, third
	31	data analysis method	*string*
	32	whether raw data is available	yes, no

A.1.3 Reporting phase

The reporting phase corresponds to plotting and reflecting on the recorded information. This includes: analyzing data from multiple perspectives, and combining information from multiple review parameters to detect patterns. Data extracted from the publications were used to answer our two research questions. From the recorded information, relevant plots, charts, and tables were produced to answer the research questions. Finally, the discussion critically reflects on the main findings to show the research gaps and guide future directions. The final results and conclusions were discussed with the study team to confirm their validity. The discussion was thoroughly reviewed by all study team members to ensure accurate interpretations. The review process also reduced the bias in the discussion.

A.2 Overview of the identified tools

Table A4 summarizes the results of our data gathering. The first column group (*i.e.*, consisting of the attributes "Study" and "Name") lists the identifiers used for the publication and tool. For each study, the corresponding tool can be identified with "T," for example for [S1] the corresponding tool is T1. Tools reported in an extended journal article as well as a previous conference paper are marked with a "*" in front of their name. The second column group lists the results corresponding to the publication trends. Specifically, "Venue" represents the venue of publication as mentioned in Table A1, "Track" represents the corresponding conference track as mentioned in section A.1.2, "Year" denotes the publication year, and "Citations" denotes the number of obtained citations (as per Google Scholar) as of February 4, 2021. Research papers are indicated with the letter "R," industry papers with "I," tool demos with "T," and journal papers with "J." The third and fourth column groups list results corresponding to the scope of the publications. The third column denotes the supported RE activity. In particular, "E" denotes elicitation, "A" indicates analysis, "S" specification, "V" validation, and "M" management. In practice, relevant modeling tools are included as *specification* tools, reasoning tools as *validation* tools, and prioritization, traceability, and monitoring tools as *management* tools. The "✠" symbol indicates RE activities that were assigned by us, in case the authors did not mention any, while a checkmark (✓) indicates claims by the authors.

The fourth column "Protot./Collab." denotes whether authors claimed their tool to be a prototype or collaborative. Finally, the last column group lists results corresponding to the availability of the tools. In particular, "Webpage" and "Source code" denote whether the URLs for the project web page and source code repositories are available. The actual available URLs for both can be found in section A.9. "Last Git activity" denotes the date of the last commit in the Git repository as of February 4, 2021. We used "n/a" in places where we lack such information.

Table A4: List of included studies

Study	Name	Venue	Track	Year	Citations	E/A/S/V/M	Protot./Collab.	Webpage	Source code	Last Git activity
[S1]	-	RE	R	2019	0	-/-/✠/-/-	✓/-	n/a	no	-
[S2]	OpenReq	RE	T	2019	0	✓/✓/-/-/-	-/-	✓	✓	10.Feb.20
[S3]	RM2PT	RE	T	2019	0	-/-/-/✓/-	-/-	n/a	✓	20.Jan.20
[S4]	CARGo	RE	T	2019	0	-/✠/✠/-/-	✓/-	n/a	✓	23.Jun.19
[S5]	T-Star	RE	T	2019	0	-/-/✠/-/-	✓/-	n/a	n/a	-
[S6]	GuideGen	RE	R	2018	0	-/-/-/-/✓	-/✓	✓	✓	11.Jan.19
[S7]	REMINDS	RE	I	2018	1	-/-/-/-/✠	-/-	✓	no	-
[S8]	-	RE	I	2018	3	-/-/✓/-/✓	-/-	n/a	n/a	-
[S9]	BloomingLeaf	RE	T	2018	4	-/✓/-/-/-	-/-	n/a	✓	23.Sep.19
[S10]	ELICA	RE	T	2018	2	✓/-/-/-/-	✓/✓	n/a	n/a	-
[S11]	FlexiView	RE	T	2018	0	-/-/✠/-/-	-/-	✓	no	-
[S12]	PiStar	RE	T	2018	9	-/-/✓/-/-	-/-	✓	✓	15.Dec.19
[S13]	SuSoftPro	RE	T	2018	5	-/✓/-/-/-	-/-	✓	no	-
[S14]	T-Reqs	RE	T	2018	5	-/-/-/-/✓	-/-	no	✓	06.Nov.19
[S15]	FAME	RE	R	2018	29	✓/-/-/-/-	-/-	n/a	✓	16.Oct.20
[S16]	SHORT	RE	R	2017	1	-/✠/✠/-/-	-/-	n/a	✓	07.Nov.16
[S17]	ASSERT™	RE	I	2017	8	-/✓/✓/-/-	-/-	n/a	n/a	-
[S18]	DMGame	RE	R	2017	2	-/-/-/-/✓	✓/✓	✓	✓	26.Apr.18
[S19]	UCAnalyzer	RE	T	2017	4	-/✓/-/-/-	✓/-	✓	✓	06.Aug.17
[S20]	RETTA	RE	T	2017	8	✓/-/-/-/-	✓/-	n/a	n/a	-
[S21]	ECrits	RE	T	2017	4	✠/-/-/-/✠	-/✓	no	no	-
[S22]	CoSTest	RE	T	2017	6	-/-/-/✓/-	✓/-	✓	no	-

Continued on next page

Table A4 – *Continued from previous page*

Study	Name	Venue	Track	Year	Citations	E/A/S/V/M	Protot./Collab.	Webpage	Source code	Last Git activity
[S23]	Canary	RE	T	2017	5	✠/-/✠/-/✠	-/-	n/a	n/a	-
[S24]	Respecify	RE	T	2017	0	-/-/✠/-/-	-/-	✓	n/a	-
[S25]	GrowingLeaf	RE	R	2016	3	-/✓/-/-/-	-/-	✓	✓	10.Oct.18
[S26]	ReqVidA	RE	R	2016	3	✓/-/-/-/-	✓/-	✓	n/a	-
[S27]	Visual Narr.	RE	R	2016	3	-/-/✠/-/-	-/-	no	✓	23.Sep.19
[S28]	PROD	RE	T	2016	4	-/-/✠/-/-	-/✓	n/a	n/a	-
[S29]	SCCMT	RE	T	2016	0	-/-/✠/-/-	-/✓	no	✓	24.Oct.14
[S30]	Capra	RE	T	2016	17	-/-/-/-/✓	✓/-	✓	✓	24.Feb.17
[S31]	MoRE	RE	T	2016	2	-/✓/-/-/-	-/-	n/a	✓	05.Sep.16
[S32]	AQUSA	RE	R	2015	4	-/-/✠/-/-	✓/-	no	✓	12.Aug.16
[S33]	Flexisketch T.	RE	R	2015	5	-/-/✠/-/-	✓/✓	✓	✓	08.May.15
[S34]	SACRE	RE	T	2015	7	-/✠/-/-/-	✓/-	✓	✓	14.Feb.18
[S35]	ReqPat	RE	T	2015	3	-/-/✓/✓/-	-/-	no	✓	16.Jul.19
[S36]	Breeze	RE	T	2015	2	-/✓/-/-/-	-/-	n/a	✓	26.Mar.15
[S37]	StakeCloud	RE	T	2015	4	✓/-/-/-/-	-/-	no	no	-
[S38]	Objectiver	RE	T	2015	6	-/✠/✠/-/-	-/-	✓	no	-
[S39]	GATO	RE	T	2015	4	-/-/✠/-/-	-/-	✓	✓	27.Aug.15
[S40]	RE-SWOT	REFSQ	R	2019	5	✓/-/-/-/-	✓/-	no	✓	13.Sep.18
[S41]	-	REFSQ	R	2018	5	-/-/✠/-/-	-/-	no	no	-
[S42]	REVV	REFSQ	R	2018	5	-/-/✠/-/-	✓/-	no	✓	03.Jan.19
[S43]	PUMConf	REFSQ	R	2017	6	-/-/✠/-/-	-/-	✓	n/a	-
[S44]	REDEPEND	REFSQ	R	2017	7	-/-/✠/-/✠	✓/-	✓	n/a	-
[S45]	Tactile Check	REFSQ	R	2017	8	-/✓/-/-/-	-/-	n/a	✓	01.Jan.17
[S46]	jUCMNav	REFSQ	R	2016	9	-/✠/✠/-/-	-/-	✓	n/a	-

Continued on next page

Table A4 – *Continued from previous page*

Study	Name	Venue	Track	Year	Citations	E/A/S/V/M	Protot./Collab.	Webpage	Source code	Last Git activity
[S47]	reqT	REFSQ	R	2016	11	-/-/✠/-/-	✓/-	n/a	✓	22.Feb.20
[S48]	-	REFSQ	R	2015	15	-/✓/-/-/-	✓/-	n/a	n/a	-
[S49]	TeAL	REFSQ	R	2015	16	-/✠/-/-/-	-/-	n/a	n/a	-
[S50]	EnLighter	ICSE	R	2019	18	-/-/✓/-/-	-/-	n/a	✓	15.Oct.18
[S51]	AutoTap	ICSE	R	2019	18	-/-/✠/-/-	✓/-	no	✓	28.Jan.19
[S52]	StoryDroid	ICSE	R	2019	19	-/-/✠/-/-	-/-	n/a	n/a	-
[S53]	Guigle	ICSE	T	2019	3	-/-/✠/-/-	-/-	✓	no	-
[S54]	iArch-U	ICSE	T	2019	0	-/-/✠/-/✠	-/-	✓	✓	29.Jan.19
[S55]	PsALM	ICSE	T	2019	6	-/-/✓/-/-	✓/-	✓	✓	04.Jul.19
[S56]	DIVER	ICSE	R	2019	1	✠/-/-/-/-	-/-	n/a	n/a	-
[S57]	IDEA	ICSE	R	2018	19	✠/-/-/-/-	-/-	n/a	✓	13.Jun.18
[S58]	GVT	ICSE	R	2018	24	-/-/-/✠/-	-/-	✓	✓	-
[S59]	-	ICSE	R	2017	28	✠/-/-/-/-	-/-	n/a	n/a	-
[S60]	RADAR	ICSE	R	2017	51	-/✓/-/-/-	-/-	✓	✓	14.Apr.19
[S61]	kEEPER	ESEC	R	2017	70	-/-/✓/-/-	✓/-	n/a	n/a	-
[S62]	NARCIA	ESEC	T	2015	8	-/✓/-/-/-	-/-	✓	n/a	-
[S63]	Prema	ASE	T	2019	0	-/✓/-/-/-	-/-	n/a	n/a	-
[S64]	BuRRiTo	ASE	T	2019	0	✠/-/-/-/-	-/-	n/a	no	-
[S65]	BProVe	ASE	T	2017	5	-/-/-/✠/-	-/✓	✓	✓	15.Mar.18
[S66]	Kobold	ASE	T	2017	8	-/-/-/✠/-	-/-	✓	✓	22.Apr.19
[S67]	AnModeler	ASE	T	2016	5	-/✓/-/-/-	-/-	✓	n/a	-
[S68]	TestMEReq	ASE	T	2016	10	-/-/-/✓/-	-/✓	no	n/a	-
[S69]	FINALIsT2	SANER	T	2018	4	✠/-/✠/-/-	-/-	no	no	-
[S70]	Scenario Ami.	RE J	J	2019	2	-/-/✓/-/-	-/-	n/a	✓	27.Jul.17

Continued on next page

Table A4 – *Continued from previous page*

Study	Name	Venue	Track	Year	Citations	E/A/S/V/M	Protot./Collab.	Webpage	Source code	Last Git activity
[S71]	ASSERT™*	RE J	J	2019	1	-/✓/✓/-/-	-/-	n/a	n/a	-
[S72]	-	RE J	J	2019	2	-/-/✓/-/-	-/-	n/a	n/a	-
[S73]	-	RE J	J	2019	6	✓/-/-/-/-	-/-	n/a	n/a	-
[S74]	-	RE J	J	2018	20	✓/-/-/-/-	-/-	n/a	n/a	-
[S75]	AMAN-DA	RE J	J	2018	7	✓/✓/-/-/-	-/-	n/a	n/a	-
[S76]	ECrits*	RE J	J	2018	2	-/✠/-/-/✠	-/-	no	no	-
[S77]	-	RE J	J	2018	1	-/-/-/-/✓	-/-	n/a	n/a	-
[S78]	CGM	RE J	J	2018	30	-/✠/✠/-/-	-/-	✓	n/a	-
[S79]	BPSTS-IQ	RE J	J	2018	8	-/✓/✓/✓/-	-/-	✓	✓	-
[S80]	MockPlug	RE J	J	2018	13	✓/✓/✓/✓/-	-/✓	no	no	-
[S81]	-	RE J	J	2017	2	-/-/-/-/✠	-/✓	n/a	n/a	-
[S82]	Visual Narr.*	RE J	J	2017	34	-/-/✠/-/-	-/-	no	✓	23.Sep.19
[S83]	LeCA	RE J	J	2017	19	-/-/✠/-/-	-/-	n/a	n/a	-
[S84]	-	RE J	J	2017	19	-/-/-/✓/-	✓/-	n/a	n/a	-
[S85]	TAOM4E	RE J	J	2017	48	-/✓/✓/-/-	-/-	✓	n/a	-
[S86]	-	RE J	J	2016	16	✓/-/-/-/-	✓/-	n/a	n/a	-
[S87]	-	RE J	J	2016	103	-/✠/-/-/-	✓/-	✓	✓	-
[S88]	AQUSA*	RE J	J	2016	95	-/-/✠/-/-	-/-	no	✓	12.Aug.16
[S89]	TiQi	RE J	J	2015	16	-/-/✠/-/-	✓/-	n/a	n/a	-
[S90]	GaiusT	RE J	J	2015	55	-/✓/-/-/-	-/-	no	no	-
[S91]	MARC	EMSE	J	2019	2	✓/-/-/-/-	✓/-	no	✓	21.Oct.18
[S92]	SNIPR	EMSE	J	2015	21	-/-/-/-/✠	✓/-	no	n/a	-
[S93]	RERD	JSS	J	2018	11	-/-/-/✓/-	-/-	✓	✓	-
[S94]	TRAILS	JSS	J	2018	5	-/-/-/-/✠	✓/-	n/a	n/a	-
[S95]	U-RUCM	JSS	J	2018	3	-/✓/✓/✓/-	-/-	✓	no	-

Continued on next page

Table A4 – *Continued from previous page*

Study	Name	Venue	Track	Year	Citations	E/A/S/V/M	Protot./Collab.	Webpage	Source code	Last Git activity
[S96]	SOVA R-TC	JSS	J	2018	2	-/✓/-/✓/-	✓/-	✓	no	-
[S97]	Smella	JSS	J	2017	68	-/-/✓/✓/-	✓/-	✓	no	-
[S98]	REVV-light*	IST	J	2019	5	-/-/✠/-/-	-/-	✓	✓	03.Jan.19
[S99]	GuideGen*	IST	J	2019	1	-/-/-/-/✓	-/-	✓	✓	11.Jan.19
[S100]	STS-IQ	IST	J	2019	0	-/✓/✓/✓/-	✓/-	✓	✓	-
[S101]	Creative Leaf	IST	J	2019	5	✓/✓/-/-/-	-/-	✓	no	-
[S102]	SREG	IST	J	2018	15	-/-/-/✠/-	-/✓	n/a	n/a	-
[S103]	UPROM	IST	J	2018	15	✓/-/✓/-/-	✓/-	✓	✓	-
[S104]	-	IST	J	2017	14	✓/-/-/-/-	-/-	n/a	n/a	-
[S105]	-	IST	J	2016	11	-/-/✠/-/-	✓/-	✓	n/a	-
[S106]	-	IST	J	2016	10	-/✓/-/-/-	✓/-	n/a	n/a	-
[S107]	Wisdom	IST	J	2016	21	-/-/-/-/✠	✓/-	n/a	n/a	-
[S108]	-	IST	J	2016	27	-/-/-/-/✓	-/-	n/a	n/a	-
[S109]	-	IST	J	2015	21	✓/✓/-/-/✓	-/-	n/a	n/a	-
[S110]	-	IST	J	2015	8	-/✠/-/-/-/	✓/-	n/a	n/a	-
[S111]	TemLoPAC	SoSyM	J	2019	6	-/-/✓/-/-	-/-	n/a	✓	17.Nov.19
[S112]	MUSER	SoSyM	J	2018	8	-/✓/-/-/-	✓/-	✓	n/a	-

A.3 SLR: threats to the validity

Several factors may have influenced the results of this study. These factors may have influenced the search, the study selection, and the extraction of the data from the selected studies.

Reliability. This concerns whether the study is reproducible by other researchers [51]. To ensure reliability, instead of relying on automatic search, we manually searched the proceedings of relevant venues. The selection process indeed leaves room for variation, as different researchers are likely to have different opinions about whether or not a publication can contribute to answering the research questions. To reduce this source of bias, the study team agreed which publications to include for analysis in this study. To reduce personal bias in selected publication assessment, at least two reviewers checked the extracted data. Also, researchers did not extract or check their own publications, and performed a pilot extraction study. Nevertheless, we only considered studies from the years 2015-2019, which can introduce some bias in the final results. We agree that several of the other SE conferences might also include RE as a topic. However, our list of conferences and journals covers the top conferences and journals the RE community uses to discuss their work. As the total number of publications included from non-RE venues (*i.e.*, everything other than RE, REFSQ, and REJ) is relatively low. Therefore, considering other non-RE venues may not change the results significantly.

Construct validity. This concerns whether the constructs are measured and interpreted correctly [51]. To ensure a common understanding of the relevant concepts and terms, we checked the relevant literature and analyzed the definitions therein. To ensure a common understanding of the data to be extracted from the studies, we performed a pilot extraction. One member extracted the data from one publication, and other study team members discussed the resulting extracted data. Although agreed by the study team, the selection and inclusion of the studies were performed primarily by the first and second author, which can introduce selection bias. Likewise, the implicit assignment of the supported RE activities to certain tools where authors failed to mention is subjective. The assignment of additional evaluation types to the studies in case authors failed to mention one is also subjective.

Internal validity. This concerns whether the study results really follow from the data [51]. Since we only use descriptive statistics in our data analysis, the threats to internal validity are minimal. The accuracy of the results and conclusions suffer from the fact that we could not collect data for all parameters for all included studies as some of the authors failed to respond to our queries. Additionally, we also recognize the fact that our analysis of performed evaluation by authors or tool types is based on the available information in the study that may lack important details due to the nature of the publication, *i.e.*, being published as a tool paper versus being published as a full research paper.

External validity. This concerns whether claims for the generality of the results are justified [51]. Kitchenham *et al.* propose four quality questions for evaluating systematic literature reviews [73]. These are presented below, along with an evaluation of our study against these questions:
QA1: Are the reviews of inclusion and exclusion criteria described and appropriate? We explicitly defined and discussed the inclusion and exclusion criteria used in item A.1.1, therefore this quality criterion is met.
QA2: Is the literature search likely to have covered all relevant studies? According to Kitchenham *et al.* this criterion is met if four or more digital libraries have been searched and additional search strategies have been included. This quality criterion is met, as instead of relying on digital libraries, we directly and manually searched nine relevant SE and RE venues.
QA3: Did the reviewers assess the quality/validity of the included studies? Instead of relying on defining quality attributes for selected studies, we selected studies that are highly relevant to answer our research questions.
QA4: Were the basic data/studies adequately described? We consider this quality criterion as met as we used a detailed data collection form for each study. The data collection form was piloted.

A.4 Artifact analysis

A.4.1 Methodology

For each artifact, we recorded attribute values across six dimensions. Table A5 provides an overview of the considered characterization dimensions. The first column denotes the dimension followed by possible values.

Table A5: Dimensions of artifact characterization

Dimension	**Possible values**
Format	Textual, Graphical, Mixed
Nature	Digital, Physical,
Contains	*other artifacts*
Helps create	*other artifacts*
SDLC phase(s) of origin	Requirements, Design, Development and Testing, Deployment and Maintenance
SDLC phase(s) of use	Requirements, Design, Development and Testing, Deployment and Maintenance

- The dimension “Format” denotes whether an artifact is purely textual, graphical, or a mixture of both. This dimension will help us decide which capabilities an IDE is required to have to model variouus artifacts.
- The dimension “Nature” denotes whether an artifact exists physically, and whether its digital counterpart is possible. This dimension will help us decide whether we can model an artifact in an IDE.
- The dimension “Contains” denotes whether an artifact can function as a container for other artifact, *e.g.*, a story map contains story cards. This dimension will help us design relationships among various artifacts.
- The dimension “Helps create” denotes whether an artifact can be used to construct another artifact, *e.g.*, sketches help create UI wireframes.
- The dimensions “SDLC phase(s) of origin” and “SDLC phase(s) of use” denote the SDLC phase from which a specific artifact originates, and is consumed in, respectively. This dimension will also help us design relationships among various artifacts.

For each selected artifact, we manually assigned values for analysis parameters.

A.4.2 List of artifacts

Table A6: List of artifacts

#	Category	Artifact	Physical	Textual	Graphical	Mixed	Source
1	Modeling artefacts	UML diagram	-	-	-	✓	[114]
2		Domain model	-	-	-	✓	[114]
3		Process model	-	-	-	✓	[113]
4		Use-case description	-	✓	-	-	[12]
5		DSL	-	-	-	✓	[12]
6		Data flow diagram	-	-	-	✓	[12]
7		Flow chart	-	-	-	✓	[12]
8		Network diagram	-	-	-	✓	[12]
9		Robustness diagram	-	-	-	✓	[12]
10		Structure diagram	-	-	-	✓	[12]
11		User interface model	-	-	-	✓	[12]
12		System model	-	-	-	✓	[113]
13		Data model	-	-	-	✓	[12, 124]
14		Role model	-	-	-	✓	[114]
15		User model	-	-	-	✓	[12]
16		Business process model	-	-	-	✓	[12]
17		CRC card	✓	-	-	✓	[12]
18		Task model	-	-	-	✓	[114, 113]
19	Testing artefacts	Test specification	-	✓	-	-	[114]
20		Integration test	-	✓	-	-	[16]
21		Regression test	-	✓	-	-	[16]
22		Unit test	-	✓	-	-	[16]
23		Test plan	-	✓	-	-	[16]
24		Scenario	-	✓	-	-	[12]
25	Maps	Mind map	-	-	-	✓	[114, 113]
26		Roadmap	-	-	-	✓	[114, 113]
27		Impact map	-	-	-	✓	[114, 113]
28		Effect map	-	-	-	✓	[114]

Continued on next page

Table A6 – *Continued from previous page*

#	Category	Artifact	Physical	Textual	Graphical	Mixed	Source
29		Story map	✓	-	-	✓	[114, 113]
30		Vision	-	-	-	✓	[12, 124, 114]
31		Design concept	-	-	-	✓	[114]
32		Business rule definition	-	✓	-	-	[12]
33		Change case	-	✓	-	-	[12]
34		Task	-	✓	-	-	[114, 113]
35	Development artefacts	Persona	✓	-	-	✓	[114, 113]
36		Issue	-	✓	-	-	[12, 124, 114, 113, 16]
37		Constraint definition	-	✓	-	-	[12]
38		External interface specification	-	✓	-	-	[12]
39		Prototype	-	-	-	✓	[12, 114, 113]
40		Mockup	-	-	-	✓	[12, 114]
41		Sketch	✓	-	-	✓	[12, 114]
42		Wireframe	-	-	-	✓	[124, 114]
43	End-user related artefacts	Interaction scenario	-	-	-	✓	[114]
44		Minimum Viable Product	-	-	-	✓	[113, 16]
45		User journey	-	-	-	✓	[114, 113]
46		User wish list	-	✓	-	-	[114]
47		Kanban board	✓	-	-	✓	[114, 113, 16]
48		Release plan	-	-	-	✓	[12, 16]
49		Story estimate	-	✓	-	-	[16]
50		Product backlog	-	✓	-	-	[124, 113, 16]
51	Project management artefacts	Burn down chart	-	-	-	✓	[16]
52		Tag	-	✓	-	-	[114]
53		Organization chart	-	-	-	✓	[12]
54		Storyboard	✓	-	-	✓	[12, 114, 113]
55		Glossary	-	✓	-	-	[12]
56		User stories	-	✓	-	-	[12, 124, 114, 113, 16]
57		Story card	✓	-	-	✓	[12, 113, 16]
58		Post-it notes	✓	-	-	✓	[12]
59	Requirements related artefacts	Epic	-	✓	-	-	[124, 114, 113]

Continued on next page

Table A6 – *Continued from previous page*

#	Category	Artifact	Physical	Textual	Graphical	Mixed	Source
60		Feature	-	✓	-	-	[12, 124, 114, 113, 16]
61		Picture	✓	-	✓	-	[12, 114, 113]
62		Video	-	-	✓	-	[114, 113]

A.5 The survey instrument

A.5.1 Interviewee background

1. Do you have previous experience working in a team on a software project? (*Yes, No*)

2. How many years of experience do you have with software development or working in a team on a project? (*free text answer*)

3. Do you have prior knowledge of agile development methodologies or previous experience with the same? (*Yes, No*)

4. How many years of experience do you have with agile development? (*free text answer*)

5. Have you previously used several tools (intended to accomplish a certain task) as a part of a toolchain in a software project? (*Yes, No*)

6. While working on a project, how often do you read project or product documentation? Please provide a value from 1: very rarely to 4: very often.

7. Could you please mention some of the main reasons you read any type of documentation, and where do you read it? (*free text answer*)

8. What applies to you? (1) I have participated in research in software engineering, (2) I am a practicing researcher, (3) I am a practitioner.

9. If you are/have been a researcher, in which research communities have you been active? (*free text answer*)

A.5.2 Post demo survey

Perceived advantages of the "moldable artifacts" approach

1. In your opinion, is the "moldable artifacts" approach convenient for artifact management? Provide a value from 1: very low to 4: very high.

2. In your opinion, is the "moldable artifacts" approach promising for handling artifact traceability? Provide a value from 1: very low to 4: very high.

3. In your opinion, is the "moldable artifacts" approach promising for live documentation? Provide a value from 1: very low to 4: very high.

4. In your opinion, does the "moldable artifacts" approach reduce the number of tools required to be used in a software project? Provide a value from 1: very low to 4: very high.

5. In your opinion, does the "moldable artifacts" approach reduce the context switches between different tools to accomplish a single task? Provide a value from 1: very low to 4: very high.

6. In your opinion, does the "moldable artifacts" approach reduce the cost of maintaining various artifacts up-to-date? Provide a value from 1: very low to 4: very high.

7. In your opinion, does the "moldable artifacts" approach provide more accurate and up-to-date matrices (*e.g.*, pending workload, accomplished tasks) necessary for decision making? Provide a value from 1: very low to 4: very high.

8. In your opinion, does the "moldable artifacts" approach reduce the manual efforts required in maintaining the project documentation up-to-date? Provide a value from 1: very low to 4: very high.

9. Given that appropriate interfaces (*e.g.*, GUI) and connectivity among various artifacts exists, do you agree that instead of employing numerous tools, an IDE would be a better platform for both technical and non-technical stakeholders to manage the entire software development process? Provide a value from 1: strongly disagree to 4: strongly agree.

10. Please describe three perceived advantages of "moldable artifacts" approach. (*free text answer*)

Perceived limitations of the "moldable artifacts" approach

1. In your opinion, what are the main limitations of the "moldable artifacts" approach? (*free text answer*)

2. As a non-technical stakeholder, would you be comfortable working with an IDE, given that an interactive workflow and appropriate GUIs for artifacts creation and management exist? Provide a value from 1: strongly disagree to 4: strongly agree.

A.6 Model BDD scenarios

Let us consider the following scenarios for *User story 11*:

```
Scenario 1: Customers place an order to take away
(only milk products)
Given an empty order
When the waiter adds Cappuccino to the empty order
And a cup of Cappuccino costs 4 EUR
And a cup of Cappuccino is taxed at 7%
And the waiter generates the Invoice for the order
Then the total invoice price is 7.28 EUR

Scenario 2: Customers place an order to take away
(combination of products)
Given an empty order
When the waiter adds a Cappuccino and a black coffee
to the empty order
And a cup of Cappuccino costs 4 EUR
And a cup of black coffee costs 3 EUR
And a cup of Cappuccino is taxed at 7%
And a cup of black coffee is taxed at 19%
And the waiter generates the Invoice for the order
Then the total invoice price is 7.85 EUR

Scenario 3: Customer place an order to take away
(no milk products)
Given an empty order
When the waiter adds black coffee to the empty order
And a cup of black coffee costs 3 EUR
And a cup of black coffee is taxed at 19%
And the waiter generates the Invoice for the order
Then the total invoice price is 3.57 EUR

Scenario 4: Customer place an order to to take away
(combination of drink and food)
Given an empty order
When the waiter adds a black coffee and a pizza
margherita to the empty order
And a cup of black coffee costs 3 EUR
And pizza margherita costs 5 EUR
And a cup of black coffee is taxed at 19%
And pizza margherita is taxed at 7%
And the waiter generates the Invoice for the order
Then the total invoice price is 5.92 EUR
```

Listing 9: A sample feature description with a scenario

A.7 BDD open-source project analysis

A.7.1 Study design

The goal of this study is to investigate the characteristics of specification files in open-source projects. The study *context* consists of 23 open-source repositories hosted on GitHub, selected from a total of 50,000.

Data gathering process

For this analysis, we need a large set of projects that use Gherkin to specify application behavior. We considered open-source projects hosted on GitHub that use the Gherkin language. Figure A2 summarizes the data gathering process we followed. First, we used the GitHub search API to retrieve a list of repositories sorted by popularity. The limitation of this approach is that we must add at least one restriction for the search API to provide any results. A similar approach is applied in another study where the authors collected open-source projects hosted on GitHub ranked in

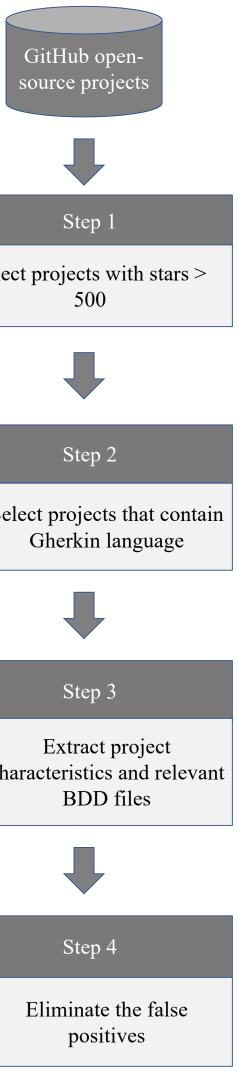

GitHub open-source projects
Step 1
Select projects with stars > 500
Step 2
Select projects that contain Gherkin language
Step 3
Extract project characteristics and relevant BDD files
Step 4
Eliminate the false positives

Figure A2: The data gathering process

terms of the number of stars [149]. We decided to retrieve only repositories with more than 500 stars, which resulted in a total of 54,277 repositories (as of 29 January 2021). This strategy allowed us to include the projects with a certain level of popularity, and enabled us to exclude projects that are not maintained. Furthermore, it allowed us to reduce the scope of the projects to be processed and allowed us to respect the permitted quota of API requests for more in-depth qualitative analysis. For all the identified

Table A7: Identified repositories according to programming languages

Language	# Repositories	Language	# Repositories
Ruby	82	Javascript	33
PHP	38	Go	16
Java	37	Emacs Lisp	11
Python	36	C#	10
C	17	TypeScript	5

repositories, we then fetched the list of programming languages. Since our interest was in the BDD projects that use Gherkin language, we selected only those that contain the Gherkin language. If repository languages contain Gherkin, then it means they contain a specification file with *.feature* extension. In this step, we retrieved a total of 318 repositories.

To analyze the specification files, we first grouped the identified repositories according to the primary programming language. As we can see from Table A7, projects with Ruby as the main programming language have used BDD the most, followed by PHP and Java. For our analysis, we focused on Java as the primary language, *i.e.*, in total 37 repositories. In these 37 repositories, we identified specification files by searching for the *.feature* extension. Additionally, we used the file search API to collect a list of files that contain the keywords *Given* and *Cucumber*. The BDD frameworks use the keyword *Given* to describe the context of the test case in BDD tools. Likewise, they use the word *Cucumber* when importing the libraries while writing the test case. We only selected those repositories that had more than one *.feature* file, *i.e.*, a total of 27 repositories, as we speculate the projects with only one *.feature* file probably only tried BDD and did not practice it rigorously. However, identifying specification files was not a trivial task as some repositories contained *.feature* files, which were not Gherkin files (*e.g.*, Torvalds and Linux). We manually inspected the specification files, but found a few false positives. To eliminate these occurrences, we compared the list of files ending with *.feature* and the list of files containing the keyword *Given*. If at least one file occurs in both list, it is very likely that the result is a true Gherkin file. Eventually, we eliminated a total of 4 false positives from these 27 repositories. We manually cross-verified the obtained results. Consequently, the process resulted in a total of 23 repositories for our qualitative analysis.

To provide the community with some meta-level information about the repositories we analyzed, we gathered a few common characteristics including: (1) number of stars, watchers, and forks, (2) number of contributors and members, and (3) number of commits, pull requests, and issues (both open & closed). We collected a total of 1,572 *.feature* files from 23 repositories. The dataset to reproduce the results can be found in the replication package.[4]

A.8 BDD tools analysis

A.8.1 Study design

The aforementioned 13 BDD tools are available as IDE plugins. We evaluated the IDE plugins according to six parameters to understand how these plugins enable specification and verification of the behavior, in other words, what opportunities non-technical stakeholders have in IDE plugins to specify and verify the application behavior.

- *Input type for specifying the scenario.* (i) "Plain text," which means a specification is written as a natural language text, (ii) "Markdown text," which means a specification is written in a Markdown format, (iii) "Table," which means a specification accepts input values in a tabular format, or (iv) "Code," meaning a specification is written in some programming language.

- *Type of parameters in the glue code.* (i) "Primitive," such as strings, numbers, or boolean values as input parameters, or (ii) "Object," which means a scenario can take domain objects as inputs.

- *Specification interface.* (i) "Textual," which means specifications can be written only as text, or (ii) "Graphical," which means specifications can be composed by using graphical elements.

- *Output type.* (i) "Test run status," which means the tool only indicates a pass or fail status for tests, or (ii) "Report," which means the tool provides alternatives to customize test run reports so that the output is readable by non-technical stakeholders.

We define our assessment parameters in subsection A.8.1. The symbol "✓" denotes that the value is "true," whereas "nc" means "not clear from the documentation."

[4]https://github.com/CodeOneTwo/software-composition-seminar

Table A8: BDD tool comparison

Tool	Input type				Parameter type		Specification interface		Output type	
	Plain text	Markdown	Table	Code	Primitive	Object	Textual	Graphical	Run Status	Report
Cucumber	✓	-	✓	-	✓	✓	✓	-	✓	✓
JBehave	✓	-	-	-	nc	nc	✓	-	✓	-
Concordion	✓	✓	✓	-	✓	✓	✓	-	✓	-
SpecFlow	✓	-	✓	-	✓	✓	✓	-	✓	✓
Spock	-	-	-	✓	✓	✓	✓	-	✓	-
RSpeck	-	-	-	✓	✓	nc	✓	-	✓	nc
MSpec	-	-	-	✓	✓	nc	✓	-	✓	-
LightBDD	-	-	-	✓	✓	nc	✓	-	✓	✓
ScalaTest	-	-	-	✓	✓	✓	✓	-	✓	-
Specs2	-	-	-	✓	✓	nc	✓	-	✓	✓
JGiven	-	-	-	✓	✓	nc	✓	-	✓	✓
phpspec	-	-	-	✓	nc	✓	✓	-	✓	✓
Gauge	-	✓	✓	-	✓	nc	✓	-	✓	✓

A.9 Included studies in the SLR

S1. Yotaro Seki, Shinpei Hayashi, and Motoshi Saeki. 2019. Detecting Bad Smells in Use Case Descriptions. In 2019 IEEE 27th International Requirements Engineering Conference (RE). IEEE, 98–108.

S2. Christoph Stanik and Walid Maalej. 2019. Requirements Intelligence with Open-Req Analytics. In 2019 IEEE 27th International Requirements Engineering Conference (RE). IEEE, 482–483. Webpage accessible at `https://openreq.eu/`, Source code repo accessible at `https://github.com/OpenReqEU`.

S3. Yilong Yang, Wei Ke, and Xiaoshan Li. 2019. RM2PT: Requirements Validation through Automatic Prototyping. In 2019 IEEE 27th International Requirements Engineering Conference (RE). IEEE, 484–485. Source code repo accessible at `https://github.com/RM2PT`.

S4. Novarun Deb, Manjarini Mallik, Anwesha Roychowdhury, and Nabendu Chaki. 2019. CARGo: A Prototype for Contextual Annotation and Reconciliation of Goal Models. In 2019 IEEE 27th International Requirements Engineering Conference (RE). IEEE, 486–489. Source code repo accessible at `https://github.com/CARGoTool/CARGoV1.0`.

S5. Yiwen Chen, Yuanpeng Wang, Yixuan Hou, and Yunduo Wang. 2019. T-Star: A Text-Based iStar Modeling Tool. In 2019 IEEE 27th International Requirements Engineering Conference (RE). IEEE, 490–491.

S6. Sofija Hotomski and Martin Glinz. 2018. A qualitative study on using GuideGento keep requirements and acceptance tests aligned. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 29–39. Webpage at `https://www.ifi.uzh.ch/en/rerg/research/GuideGen.html`, Source code repo accessible at `https://github.com/hotomski/guidegen`.

S7. Michael Vierhauser, Jane Cleland-Huang, Rick Rabiser, Thomas Krismayer, and Paul Grünbacher. 2018. Supporting diagnosis of requirements violations in systems of systems. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 325–335. Webpage accessible at `http://mevss.jku.at/?page_id=1470`.

S8. Dalton N Jorge, Patrícia DL Machado, Everton LG Alves, and Wilkerson L An-drade. 2018. Integrating Requirements Specification and Model-Based Testing in Agile Development. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 336–346.

S9. Alicia M Grubb and Marsha Chechik. 2018. Bloomingleaf: a formal tool for requirements evolution over time. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 490–491. Source code repo accessible at `https://github.com/amgrubb/BloomingLeaf`.

S10. Zahra Shakeri Hossein Abad, Munib Rahman, Abdullah Cheema, Vincenzo Gervasi, Didar Zowghi, and Ken Barker. 2018. Dynamic visual analytics for elicitation meetings with elica. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 492–493.

S11. Parisa Ghazi and Martin Glinz. 2018. FlexiView Experimental Tool: Fair and Detailed Usability Tests for Requirements Modeling Tools. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 494–495. Webpage at `https://www.ifi.uzh.ch/en/rerg/research/flexiview.html`.

S12. Joao Pimentel and Jaelson Castro. 2018. pistar tool–a pluggable online tool for goal modeling. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 498–499. Webpage accessible at `https://www.cin.ufpe.br/~jhcp/pistar/`, Source code repo accessible at `https://github.com/jhcp/pistar`.

S13. Ahmed D Alharthi, Maria Spichkova, and Margaret Hamilton. 2018. Susoftpro: Sustainability profiling for software. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 500–501. Webpage accessible at `https://ahmedalharthi.net/susoftpro/`.

S14. Eric Knauss, Grischa Liebel, Jennifer Horkoff, Rebekka Wohlrab, Rashidah Kasauli, Filip Lange, and Pierre Gildert. 2018. T-Reqs: Tool support for managing requirements in large-scale agile system development. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 502–503. Source code repo accessible at `https://github.com/regot-chalmers/treqs`.

S15. Marc Oriol, Melanie Stade, Farnaz Fotrousi, Sergi Nadal, Jovan Varga, Norbert Seyff, Alberto Abello, Xavier Franch, Jordi Marco, and Oleg Schmidt. 2018. FAME: supporting continuous requirements elicitation by combining user feedback and monitoring. In 2018 IEEE 26th International Requirements Engineering Conference (RE). IEEE, 217–227. Source code repo accessible at (CHECK) .

S16. George Mathew, Tim Menzies, Neil A Ernst, and John Klein. 2017. "SHORT" er Reasoning About Larger Requirements Models. In 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE, 154–163. Source code repo accessible at `https://github.com/dr-bigfatnoob/softgoals`.

S17. Andrew Crapo, Abha Moitra, Craig McMillan, and Daniel Russell. 2017. Requirements capture and analysis in ASSERT (TM). In 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE, 283–291.

S18. Fitsum Meshesha Kifetew, Denisse Munante, Anna Perini, Angelo Susi, Alberto Siena, Paolo Busetta, and Danilo Valerio. 2017. Gamifying collaborative prioritization: Does pointsification work?. In 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE, 322–331. Webpage accessible at `https://supersede-project.github.io/dm_game/`, Source code repo accessible at `https://github.com/supersede-project/dm_game`.

S19. Saurabh Tiwari and Mayank Laddha. 2017. UCAnalyzer: A Tool to Analyze Use Case Textual Descriptions. In 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE, 448–449. Webpage accessible at `https://sites.google.com/view/ucanalyzer/`

home?authuser=0, Source code accessible at https://github.com/maylad31/ucanalyzer.

S20. Mohammad Noaeen, Zahra Shakeri Hossein Abad, and Behrouz Homayoun Far. 2017. Let's hear it from RETTA: A Requirements Elicitation Tool for TrAffic management systems. In 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE, 450–451.

S21. Lloyd Montgomery, Emma Reading, and Daniela Damian. 2017. Ecrits—visualizing support ticket escalation risk. In 2017 IEEE 25th international requirements engineering conference (RE). IEEE, 452–455.

S22. Maria Fernanda Granda, Nelly Condori-Fernández, Tanja EJ Vos, and Oscar Pastor. 2017. CoSTest: A tool for validation of requirements at model level. In 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE, 464–467. Webpage accessible at https://costestproject2017.wordpress.com/.

S23. Georgi M Kanchev, Pradeep K Murukannaiah, Amit K Chopra, and Pete Sawyer. 2017. Canary: an interactive and query-based approach to extract requirements from online forums. In 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE, 470–471.

S24. Michael Ledger. 2017. A demonstration of Respecify: a requirements authoring tool harnessing CNL. In 2017 IEEE 25th International Requirements Engineering Conference (RE). IEEE, 472–473. Webpage accessible at http://quasimal.com/projects/respecify.html.

S25. Alicia M Grubb and Marsha Chechik. 2016. Looking into the crystal ball: requirements evolution over time. In 2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE, 86–95. Webpage accessible at http://www.cs.utoronto.ca/~amgrubb/growing-leaf/, Source code repo accessible at https://github.com/amgrubb/GrowingLeaf.

S26. Oliver Karras, Stephan Kiesling, and Kurt Schneider. 2016. Supporting requirements elicitation by
tool-supported video analysis. In 2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE, 146–155. Webpage accessible at http://www.se.uni-hannover.de/pages/en:projekte_reqvida.

S27. Marcel Robeer, Garm Lucassen, Jan Martijn EM van der Werf, Fabiano Dalpiaz, and Sjaak Brinkkemper. 2016. Automated extraction of conceptual models from user stories via NLP. In 2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE, 196–205. Source code repo accessible at https://github.com/MarcelRobeer/VisualNarrator.

S28. Ning Gao and Zhi Li. 2016. Generating testing codes for behavior-driven development from problem diagrams: A tool-based approach. In 2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE, 399–400.

S29. Yi Jiang, Shijun Wang, Kai Fu, Wei Zhang, and Haiyan Zhao. 2016. SCCMT: AStigmergy-Based Collaborative Conceptual Modeling Tool. In

2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE, 401–404. Source code repo at `https://github.com/Jexceed/Stigmergic-modeling`.

S30. Salome Maro and Jan-Philipp Steghöfer. 2016. Capra: A configurable and extendable traceability management tool. In 2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE, 407–408. Webpage accessible at `https://projects.eclipse.org/projects/modeling.capra`, Source code repo accessible at `https://bit.ly/3JTu8Z4`.

S31. Xiaozhou Li, Biswa Upreti, and Zheying Zhang. 2016. Mobility Requirements Engineering Tool (MoRE). In 2016 IEEE 24th International Requirements Engineering Conference (RE). IEEE, 409–410. Source code repo accessible at `https://github.com/biswaupreti/MobilityRequirements`.

S32. Garm Lucassen, Fabiano Dalpiaz, Jan Martijn EM van der Werf, and Sjaak Brinkkemper. 2015. Forging high-quality user stories: towards a discipline for agile requirements. In 2015 IEEE 23rd international requirements engineering conference (RE). IEEE, 126–135. Source code repo accessible at `https://github.com/gglucass/aqusa`.

S33. Dustin Wüest, Norbert Seyff, and Martin Glinz. 2015. Sketching and notation creation with FlexiSketch Team: Evaluating a new means for collaborative requirements elicitation. In 2015 IEEE 23rd International Requirements Engineering Conference (RE). IEEE, 186–195. Webpage at `https://www.ifi.uzh.ch/en/rerg/research/flexiblemodeling/flexisketch.html`, Source code repo at `https://files.ifi.uzh.ch/rerg/flexisketch/ICSE2015/`.

S34. Edith Zavala, Xavier Franch, Jordi Marco, Alessia Knauss, and Daniela Damian. 2015. SACRE: a tool for dealing with uncertainty in contextual requirements at runtime. In 2015 IEEE 23rd International Requirements Engineering Conference(RE). IEEE, 278–279. Webpage accessible at `https://gessi.upc.edu/en/tools/sacre-tool/sacre`, Source code repo accessible at `https://github.com/edithzavala/sacre-sv`.

S35. Markus Fockel and Jörg Holtmann. 2015. ReqPat: Efficient documentation of high-quality requirements using controlled natural language. In 2015 IEEE 23rd International Requirements Engineering Conference (RE). IEEE, 280–281. Source code repo accessible at `https://github.com/fraunhofer-iem/reqpat`.

S36. Luxi Chen, Linpeng Huang, Hao Zhong, Chen Li, and Xiwen Wu. 2015. Breeze: A modeling tool for designing, analyzing, and improving software architecture. In 2015 IEEE 23rd International Requirements Engineering Conference (RE). IEEE,284–285. Source code repo accessible at `https://github.com/BreezeCSA/Breeze`.

S37. Irina Todoran Koitz and Martin Glinz. 2015. StakeCloud Tool: From cloud consumers' search queries to new service requirements. In 2015 IEEE 23rd International Requirements Engineering Conference (RE). IEEE, 286–287.

S38. Robert Darimont and Christophe Ponsard. 2015. Supporting quantitative assessment of requirements in goal orientation. In 2015 IEEE 23rd International Requirements Engineering Conference (RE). IEEE, 290–291. Webpage accessible at `http://www.objectiver.com/index.php?id=4`.

S39. João Pimentel, Jéssyka Vilela, and Jaelson Castro. 2015. Web tool for goal modelling and statechart derivation. In 2015 IEEE 23rd International Requirements Engineering Conference (RE). IEEE, 292–293. Webpage accessible at `https://www.cin.ufpe.br/~jhcp/gato/about.html`, Source code repo accessible at `https://github.com/jhcp/GoalArch`.

S40. Fabiano Dalpiaz and Micaela Parente. 2019. RE-SWOT: From User Feedback to Requirements via Competitor Analysis. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer, 55–70. Source code repo accessible at `https://github.com/RELabUU/RE-SWOT`.

S41. Jonas Paul Winkler and Andreas Vogelsang. 2018. Using tools to assist identification of non-requirements in requirements specifications– A controlled experiment. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer, 57–71.

S42. Fabiano Dalpiaz, Ivor Van der Schalk, and Garm Lucassen. 2018. Pinpointing ambiguity and incompleteness in requirements engineering via information visualization and NLP. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer, 119–135. Source code repo accessible at `https://github.com/RELabUU/revv-light` and `https://github.com/RELabUU/revv`.

S43. Ines Hajri, Arda Goknil, Lionel C Briand, and Thierry Stephany. 2017. Incremental reconfiguration of product specific use case models for evolving configuration decisions. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer, 3–21. Webpage accessible at `https://sites.google.com/site/pumconf/`.

S44. James Lockerbie, Neil Maiden, Chris Williams, and Leigh Chase. 2017. A Requirements Traceability Approach to Support Mission Assurance and Configurability in the Military. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer, 308–323. Webpage at `https://bit.ly/33gKutV`.

S45. Martin Wilmink and Christoph Bockisch. 2017. On the ability of lightweight checks to detect ambiguity in requirements documentation. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer, 327–343. Source code repo accessible at `https://github.com/mwmk67/TactileCheck`.

S46. Qin Ma and Sybren de Kinderen. 2016. Goal-based decision making. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer, 19–35. Webpage accessible at `http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome`.

S47. Björn Regnell. 2016. What is essential?– A pilot survey on views about the requirements metamodel of reqT. org. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer, 232–239. Source code repo accessible at `https://github.com/reqT/reqT`.

S48. Tong Li, Jennifer Horkoff, and John Mylopoulos. 2015. Analyzing and enforcing security mechanisms on requirements specifications. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer,115–131.

S49. Wenbin Li, Jane Huffman Hayes, and Mirosław Truszczyński. 2015. Towards More Efficient Requirements Formalization: A Study. In International Working Conference on Requirements Engineering: Foundation for Software Quality. Springer,181–197.

S50. Sami Lazreg, Maxime Cordy, Philippe Collet, Patrick Heymans, and Sébastien Mosser. 2019. Multifaceted automated analyses for variability-intensive embedded systems. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 854–865. Source code repo accessible at `https://bitbucket.org/SamiLazreg/enlighter/src/master/`.

S51. Lefan Zhang, Weijia He, Jesse Martinez, Noah Brackenbury, Shan Lu, and Blase Ur. 2019. AutoTap: synthesizing and repairing trigger-action programs using LTL properties. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 281–291. Source code repo accessible at `https://github.com/zlfben/autotap`.

S52. Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. Storydroid: Automated generation of storyboard for Android apps. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 596–607.

S53. Carlos Bernal-Cárdenas, Kevin Moran, Michele Tufano, Zichang Liu, Linyong Nan, Zhehan Shi, and Denys Poshyvanyk. 2019. Guigle: a GUI search engine for Android apps. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, 71–74. Webpage accessible at `http://www.guigle.com/`.

S54. Naoyasu Ubayashi, Takuya Watanabe, Yasutaka Kamei, and Ryosuke Sato. 2019. Git-based integrated uncertainty manager. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, 95–98. Webpage accessible at `http://posl.github.io/iArch/`, Source code repo accessible at `https://github.com/posl/iArch`.

S55. Claudio Menghi, Christos Tsigkanos, Thorsten Berger, and Patrizio Pelliccione. 2019. PsALM: specification of dependable robotic missions. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). IEEE, 99–102. Webpage at `http://roboticpatterns.com/psalm/`, Source code repo at `https://github.com/claudiomenghi/PsAlM`.

S56. Cuiyun Gao, Wujie Zheng, Yuetang Deng, David Lo, Jichuan Zeng, Michael R Lyu, and Irwin King. 2019. Emerging app issue identification from user feedback: experience on WeChat. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 279–288.

S57. Cuiyun Gao, Jichuan Zeng, Michael R Lyu, and Irwin King. 2018. Online app review analysis for identifying emerging issues. In Proceedings of the 40th International Conference on Software Engineering. 48–58. Source code repo accessible at `https://github.com/armor-ai/IDEA`.

S58. Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated reporting of GUI design violations for mobile apps. In Proceedings of the 40th International Conference on Software Engineering. 165–175. Webpage accessible at `https://www.android-dev-tools.com/gvt`, Source code repo accessible at `https://www.dropbox.com/s/7yuarlbinvgj6ck/GVT-SRC-ICSE18.tar.gz?dl=0`.

S59. Paige Rodeghero, Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Detecting user story information in developer-client conversations to generate extractive summaries. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 49–59.

S60. Saheed A Busari and Emmanuel Letier. 2017. Radar: A lightweight tool for requirements and architecture decision analysis. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 552–562. Webpage accessible at `https://ucl-badass.github.io/radar/`, Source code repo accessible at `https://github.com/sbusari/RADAR`.

S61. Dalal Alrajeh, Liliana Pasquale, and Bashar Nuseibeh. 2017. On evidence preservation requirements for forensic-ready systems. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 559–569.

S62. Chetan Arora, Mehrdad Sabetzadeh, Arda Goknil, Lionel C Briand, and Frank Zimmer. 2015. NARCIA: an automated tool for change impact analysis in natural language requirements. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. 962–965. Webpage accessible at `https://sites.google.com/site/svvnarcia/`.

S63. Yihao Huang, Jincao Feng, Hanyue Zheng, Jiayi Zhu, Shang Wang, Siyuan Jiang, Weikai Miao, and Geguang Pu. 2019. Prema: A Tool for Precise Requirements Editing, Modeling and Analysis. In 2019 34th IEEE/ACM International Conferenceon Automated Software Engineering (ASE). IEEE, 1166–1169.

S64. Pavan Kumar Chittimalli, Kritika Anand, Shrishti Pradhan, Sayandeep Mitra, Chandan Prakash, Rohit Shere, and Ravindra Naik. 2019. BuRRiTo: A Framework to Extract, Specify, Verify and Analyze Business Rules. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 1190–1193.

S65. Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, Francesco Tiezzi, and Andrea Vandin. 2017. BProVe: tool support for business process verification. In 2017 32nd IEEE/ACM International Conference on

Automated Software Engineering (ASE). IEEE, 937–942. Webpage accessible at `http://pros.unicam.it/bprove/`, Source code repo accessible at `https://bitbucket.org/proslabteam/bprove/src/master/`.

S66. Julián Grigera, Alejandra Garrido, and Gustavo Rossi. 2017. Kobold: web usability as a service. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 990–995. Webpage accessible at `http://autorefactoring.lifia.info.unlp.edu.ar/`, Source code repo accessible at `https://github.com/juliangrigera/Kobold`.

S67. Jitendra Singh Thakur and Atul Gupta. 2016. AnModeler: a tool for generating domain models from textual specifications. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 828–833. Webpage accessible at `https://sites.google.com/site/anmodeler/`.

S68. Nor Aiza Moketar, Massila Kamalrudin, Safiah Sidek, Mark Robinson, and John Grundy. 2016. An automated collaborative requirements engineering tool for better validation of requirements. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 864–869.

S69. Andreas Burger and Sten Grüner. 2018. Finalist 2: Feature identification, localization, and tracing tool. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 532–537.

S70. Deokyoon Ko, Suntae Kim, and Sooyong Park. 2019. Automatic recommendation to omitted steps in use case specification. Requirements Engineering 24, 4 (2019), 431–458. Source code repo accessible at `https://github.com/maniara/ScenarioAmigo`.

S71. Abha Moitra, Kit Siu, Andrew W Crapo, Michael Durling, Meng Li, Panagiotis Manolios, Michael Meiners, and Craig McMillan. 2019. Automating requirements analysis and test case generation. Requirements Engineering 24, 3 (2019), 341–364.

S72. Jonas Westman and Mattias Nyberg. 2019. Providing tool support for specifying safety-critical systems by enforcing syntactic contract conditions. RequirementsEngineering 24, 2 (2019), 231–256.

S73. Deepika Prakash and Naveen Prakash. 2019. A multifactor approach for elicitation of Information requirements of data warehouses. Requirements Engineering 24, 1(2019), 103–117.

S74. Sarah Thew and Alistair Sutcliffe. 2018. Value-based requirements engineering: method and experience. Requirements engineering 23, 4 (2018), 443–464.

S75. Amina Souag, Raúl Mazo, Camille Salinesi, and Isabelle Comyn-Wattiau. 2018. Using the AMAN-DA method to generate security requirements: a case study in the maritime domain. Requirements Engineering 23, 4 (2018), 557–580.

S76. Lloyd Montgomery, Daniela Damian, Tyson Bulmer, and Shaikh Quader. 2018. Customer support ticket escalation prediction using feature engineering. Requirements Engineering 23, 3 (2018), 333–355.

S77. Shinobu Saito, Yukako Iimura, Aaron K Massey, and Annie I Antón. 2018. Discovering undocumented knowledge through visualization of agile software development activities. Requirements Engineering 23, 3 (2018), 381–399.

S78. Chi Mai Nguyen, Roberto Sebastiani, Paolo Giorgini, and John Mylopoulos. 2018. Multi-objective reasoning with constrained goal models. Requirements Engineering 23, 2 (2018), 189–225. Webpage accessible at `http://www.cgm-tool.eu/index.html`.

S79. Mohamad Gharib, Paolo Giorgini, and John Mylopoulos. 2018. Analysis of information quality requirements in business processes, revisited. Requirements Engineering 23, 2 (2018), 227–249.
Webpage at `https://mohamadgharib.wordpress.com/bpsts-iq-tool/`, Source code at `https://bit.ly/2AAdbnA`.

S80. Diego Firmenich, Sergio Firmenich, José Matías Rivero, Leandro Antonelli, and Gustavo Rossi. 2018. CrowdMock: an approach for defining and evolving web augmentation requirements. Requirements Engineering 23, 1 (2018), 33–61.

S81. Ricardo Eito-Brun and Antonio Amescua. 2017. Dealing with software process requirements complexity: an information access proposal based on semantic technologies. Requirements Engineering 22, 4 (2017), 527–542.

S82. Garm Lucassen, Marcel Robeer, Fabiano Dalpiaz, Jan Martijn EM van der Werf, and Sjaak Brinkkemper. 2017. Extracting conceptual models from user stories with Visual Narrator. Requirements Engineering 22, 3 (2017), 339–358. Source code available at `https://github.com/MarcelRobeer/VisualNarrator`.

S83. Nicolas Sannier, Morayo Adedjouma, Mehrdad Sabetzadeh, and Lionel Briand. 2017. An automated framework for detection and resolution of cross referencesin legal texts. Requirements Engineering 22, 2 (2017), 215–237.

S84. Mohamed Amine Beghoura, Abdelhak Boubetra, and Abdallah Boukerram. 2017. Green software requirements and measurement: random decision forests-based software energy consumption profiling. Requirements Engineering 22, 1 (2017), 27–40.

S85. Mirko Morandini, Loris Penserini, Anna Perini, and Alessandro Marchetto. 2017. Engineering requirements for adaptive systems. Requirements Engineering 22, 1(2017), 77–103. Webpage accessible at `http://se.fbk.eu/technologies/taom4e`.

S86. Luz María Priego-Roche, Dominique Rieu, et al. 2016. A framework for virtual organization requirements. Requirements Engineering 21, 4 (2016), 439–460.

S87. Walid Maalej, Zijad Kurtanović, Hadeer Nabil, and Christoph Stanik. 2016. On the automatic classification of app reviews. Requirements Engineering 21, 3 (2016), 311–331. Webpage accessible at `https://mast.`

informatik.uni-hamburg.de/app-review-analysis/,
Source code available at https://mast.informatik.uni-hamburg.de/wp-content/uploads/2014/03/ReviewClassifier4J.zip.

S88. Garm Lucassen, Fabiano Dalpiaz, Jan Martijn EM van der Werf, and Sjaak Brinkkemper. 2016. Improving agile requirements: the quality user story framework and tool. Requirements Engineering 21, 3 (2016), 383–403. Source code available at https://github.com/gglucass/aqusa.

S89. Piotr Pruski, Sugandha Lohar, William Goss, Alexander Rasin, and Jane Cleland-Huang. 2015. TiQi: answering unstructured natural language trace queries. Requirements Engineering 20, 3 (2015), 215–232.

S90. Nicola Zeni, Nadzeya Kiyavitskaya, Luisa Mich, James R Cordy, and John My-lopoulos. 2015. GaiusT: supporting the extraction of rights and obligations for regulatory compliance. Requirements engineering 20, 1 (2015), 1–22.

S91. Nishant Jha and Anas Mahmoud. 2019. Mining non-functional requirements from App store reviews. Empirical Software Engineering 24, 6 (2019), 3659–3695. Source code available at https://github.com/seelprojects/MARC-3.0.

S92. Jason McZara, Shahryar Sarkani, Thomas Holzer, and Timothy Eveleigh. 2015. Software requirements prioritization and selection using linguistic tools and constraint solvers— a controlled experiment. Empirical Software Engineering 20,6 (2015), 1721–1761.

S93. Emmanouela Stachtiari, Anastasia Mavridou, Panagiotis Katsaros, Simon Bliudze, and Joseph Sifakis. 2018. Early validation of system requirements and design through correctness-by-construction. Journal of Systems and Software 145 (2018), 52–78. Webpage accessible at https://emmastac.github.io/RERD-tool/, Source code available at https://github.com/emmastac/RERD-tool.

S94. Thomas Wolfenstetter, Mohammad R Basirati, Markus Böhm, and Helmut Krcmar. 2018. Introducing TRAILS: A tool supporting traceability, integration and visualisation of engineering knowledge for product service systems development. Journal of Systems and Software 144 (2018), 342–355.

S95. Man Zhang, Tao Yue, Shaukat Ali, Bran Selic, Oscar Okariz, Roland Norgre, and Karmele Intxausti. 2018. Specifying uncertainty in use case models. Journal of Systems and Software 144 (2018), 573–603. Webpage accessible at http://zen-tools.com/rucm/U_RUCM.html.

S96. Michal Steinberger, Iris Reinhartz-Berger, and Amir Tomer. 2018. Cross lifecycle variability analysis: Utilizing requirements and testing artifacts. Journal of Systems and Software 143 (2018), 208–230. Webpage accessible at https://sites.google.com/is.haifa.ac.il/sova.

S97. Henning Femmer, Daniel Méndez Fernández, Stefan Wagner, and Sebastian Eder. 2017. Rapid quality assurance with requirements smells. Journal of Systems and Software123 (2017), 190–213. Webpage accessible at https://www.qualicen.de/?page_id=8949.

S98. Fabiano Dalpiaz, Ivor Van Der Schalk, Sjaak Brinkkemper, Fatma Başak Aydemir, and Garm Lucassen. 2019. Detecting terminological ambiguity in user stories: tool and experimentation. Information and Software Technology 110 (2019), 3–16. Webpage at `http://www.staff.science.uu.nl/~dalpi001/revv-light/`,
Source code at `https://github.com/RELabUU/revv-light`.

S99. Sofija Hotomski and Martin Glinz. 2019. GuideGen: An approach for keeping requirements and acceptance tests aligned via automatically generated guidance. Information and Software Technology 110 (2019), 17–38. Webpage accessible at `https://www.ifi.uzh.ch/en/rerg/research/GuideGen.html`, Source code accessible at `https://github.com/hotomski/guidegen`.

S100. Mohamad Gharib and Paolo Giorgini. 2019. Information quality requirements engineering with STS-IQ. Information and Software Technology 107 (2019), 83–100. Webpage accessible at `https://bit.ly/3GdCXuF`.

S101. Jennifer Horkoff, NA Maiden, and David Asboth. 2019. Creative goal modeling for innovative requirements. Information and software Technology 106 (2019), 85–100. Webpage at `https://bit.ly/3JUrzpz`.

S102. Affan Yasin, Lin Liu, Tong Li, Jianmin Wang, and Didar Zowghi. 2018. Designand preliminary evaluation of a cyber Security Requirements Education Game(SREG). Information and Software Technology 95 (2018), 179–200.

S103. Banu Aysolmaz, Henrik Leopold, Hajo A Reijers, and Onur Demirörs. 2018. A semi-automated approach for generating natural language requirements documents based on business process models. Information and Software Technology 93 (2018), 14–29. Webpage accessible at `http://www.bflow.org/uprom.html`,
Source code accessible at `http://aysolmaz.com/PrcModReqGenTool.rar`.

S104. Sangeeta Dey and Seok-Won Lee. 2017. REASSURE: Requirements elicitation for adaptive socio-technical systems using repertory grid. Information and SoftwareTechnology 87 (2017), 160–179.

S105. George Chatzikonstantinou and Kostas Kontogiannis. 2016. Run-time requirements verification for reconfigurable systems. Information and Software Technology 75 (2016), 105–121. Webpage accessible at `http://www.softlab.ntua.gr/~gechatz/seb/`.

S106. Janardan Misra. 2016. Terminological inconsistency analysis of natural language requirements. Information and Software Technology 74 (2016), 183–193.

S107. Selami Bagriyanik and Adem Karahoca. 2016. Automated COSMIC Function Point measurement using a requirements engineering ontology. Information and Software Technology 72 (2016), 189–203.

S108. Naveed Ali and Richard Lai. 2016. A method of requirements change management for global software development. Information and Software Technology 70 (2016), 49–67.

S109. Geri Georg, Gunter Mussbacher, Daniel Amyot, Dorina Petriu, Lucy Troup, Saul Lozano-Fuentes, and Robert France. 2015. Synergy between Activity Theory and goal/scenario modeling for requirements elicitation, analysis, and evolution. Information and Software Technology 59 (2015), 109–135.

S110. M Brian Blake, Iman Saleh, Yi Wei, Ian D Schlesinger, Alexander Yale-Loehr, and Xuanzhe Liu. 2015. Shared service recommendations from requirement specifications: A hybrid syntactic and semantic toolkit. Information and Software Technology 57 (2015), 392–404.

S111. Aamir M Khan, Frédéric Mallet, and Muhammad Rashid. 2019. A framework to specify system requirements using natural interpretation of UML/MARTE diagrams. Software & Systems Modeling 18, 1 (2019), 11–37. Source code accessible at `https://github.com/jadoonengr/TemLoPAC`.

S112. Tong Li, Jennifer Horkoff, and John Mylopoulos. 2018. Holistic security requirements analysis for socio-technical systems. Software & Systems Modeling 17, 4(2018), 1253–1285. Webpage accessible at `http://disi.unitn.it/~li/MUSER/Intro.html`.