# First-class artifacts as building blocks for live in-IDE documentation

Nitish Patkar*, Andrei Chis†, Nataliia Stulova*, and Oscar Nierstrasz*
*University of Bern, Switzerland
Email: http://scg.unibe.ch/staff
†Feenk GmbH, Switzerland
Email: chisvasileandrei@gmail.com

***Abstract*—A traditional round-trip engineering approach based on model transformations does not scale well to modern agile development environments where numerous artifacts are produced using a range of heterogeneous tools and technologies. To boost artifact connectivity and maintain their consistency, we propose to create and manage software-related artifacts as first-class entities directly in an integrated development environment (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 document-artifacts. We detail and exemplify our approach in the Glamorous Toolkit IDE (henceforth, Glamorous toolkit), and discuss the results of a semi-structured pilot survey we conducted with practitioners and researchers to evaluate its usefulness in practice.**

***Index Terms*—Requirements engineering, Development Environments, Software Artifacts**

## I. Introduction

A multitude of artifacts are used for distinct tasks in software development, as well as in requirements engineering [1], [2], [3], [4]. There are, for example, design artifacts [5], [6], requirements artifacts [7], and software artifacts [8]. In practice, when the requirements change, due to an abundance of employed tools for artifact creation, management, and the source code implementation: (i) establishing traceability among various artifacts and the source code becomes difficult [9], [7], and (ii) maintaining project documentation up-to-date becomes difficult [10], [11].

Kiandl and Glinz independently envisioned representing and organizing project requirements as run-time objects [12], [13]. With their approach, one classifies and organizes requirements using classes and enjoys various benefits of object-oriented design. In this work, we propose that not only requirements, but also other software-related artifacts should be created as first-class entities directly in an IDE. This approach has the following benefits:

1) all project-relevant artifacts are created and maintained directly within a development platform.No additional recovery of trace links between artifacts is required; all necessary artifacts are part of the same development infrastructure and connected to each other.
2) artifacts can be used to create stakeholder-specific interactive and executable document-artifacts.

To support this kind of software development, IDEs need to provide basic functionality to build interactive visual components for artifact representation, which would include both text and graphics. With our proposed approach, developers first create appropriate classes for the artifacts and model artifact behavior in the class methods. For instance, a user story can be modeled[1] 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. This underlying infrastructure to enable artifact object creation becomes a part of the software project source code. Other (non-technical) stakeholders create specific instances of user stories as first-class entities (*i.e.,* run-time objects) in an IDE. This workflow applies for any artifact one wishes to model in an IDE. Depending on the type of artifact, some artifacts can be connected directly to the project source code. For example, a particular user story, *e.g.,*`As a user I want to login` can be connected to the the `User` class so that one can navigate freely from a user story (*i.e.,* requirement) to the domain concept (*i.e.,* implementation) and vice versa. Later, various aspects of the running system, such as requirements or algorithms, are combined into various live document-artifacts by composing existing in-IDE artifact objects. Such live documents can be targeted towards different stakeholders. Our approach requires developers to invest once in building artifacts from scratch. Once such infrastructure exists, it can be used for any future project.

We show the feasibility of our approach by presenting three artifacts in Glamorous toolkit [14]. Glamorous toolkit is built using a graphical framework that supports the creation of customizable graphical interfaces for various types of objects. An object inspector in Glamorous toolkit allows an object to define a set of multiple interchangeable graphical presentations (*i.e., views*) capturing interesting aspects of that object in various development contexts [15]. Glamorous toolkit is a reflective environment that allows its users to query the run-time system. With such necessary infrastructure already available, it is the right fit to exemplify our approach. We will reuse the capabilities of Glamorous toolkit to support the creation, exploration, and maintenance of a variety of

[1]Please refer to the supporting material to explore a sample implementation of a user story model https://figshare.com/s/fdb27fb82544ba07ae6d

artifacts. Specifically, we use the object inspector views and its visualization engine to show specific details of an artifact to different users. In future, the same views can be used to facilitate artifact creation.

## II. Artifact Modeling

First-class artifacts are live objects that are readily available for further computation, *e.g.,* they might be passed as an argument, returned from a function, modified, and assigned to a variable. This allows artifacts to be seen as building blocks for creating other, more complex artifacts. For example, live user story objects can be used to build an in-IDE live Kanban board. Furthermore, artifacts can be associated with custom views to provide both implementation details and metadata at different abstraction levels about the artifact itself: who created this artifact, when was it created, when was it last updated, where it is used. Such information could help in maintaining multiple versions of artifacts, and proves valuable during project management. In this section, we discuss first-class implementations of three representative artifacts that support their users in distinct software development tasks.

### A. Running example

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, the development team 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.* The implementation of the running example and the following artifacts can be explored by following the instructions provided in the readme file in the additional supporting material.

### B. 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 [16]. User stories are a nice fit for the IT company to collect and specify requirements together with the hospital staff.

Due to the page limitation, in the supporting material we describe how to explore two representations of user stories in our running example implementation. 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.

### C. Mindmap

Mindmapping is a visual way of organizing and representing information within a radial hierarchy [17]. 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 management domain. A mindmap of domain concepts from the scheduling application could assist a new developer in understanding the main concepts.

In Figure 1, we show a mindmap of all major concepts in the hospital scheduling domain. Each node represents a domain concept and nodes are connected with arrows. Each node is a clickable first-class entity, thereby allowing a user to jump into the implementation of a specific concept. In Figure 1, a user has clicked on a node "HospitalSystem," and an object inspector window on the right-hand side shows the class comment for the "HospitalSystem" class, which allows the new developer to understand the implementation of each domain concept in an iterative and interactive manner. Note that from the tab "Related Stories," it is also possible to explore the related user stories (*i.e.,* requirements) for a specific domain concept, which fosters two-way connectivity between two artifacts.

### D. Scenario

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. Behavior-driven scenarios are written using the Gherkin language. Frameworks like Cucumber even make scenarios executable [18].

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 2, we show an executable scenario written in Pharo 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 examples" 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

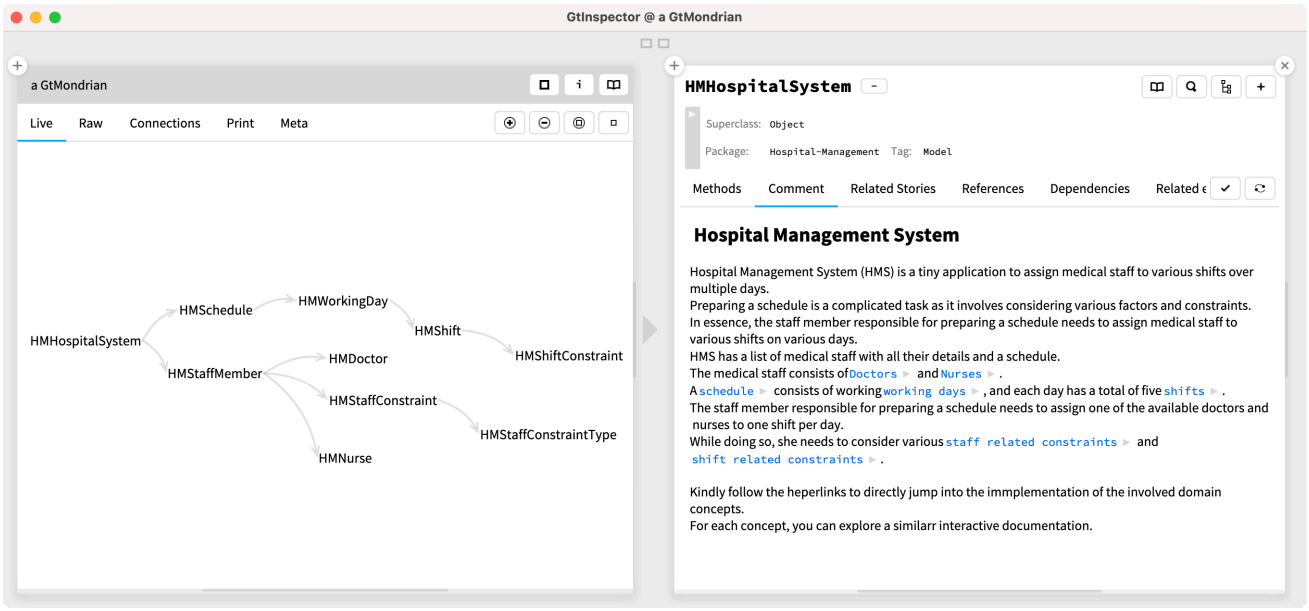


Fig. 1: A sample in-IDE mindmap

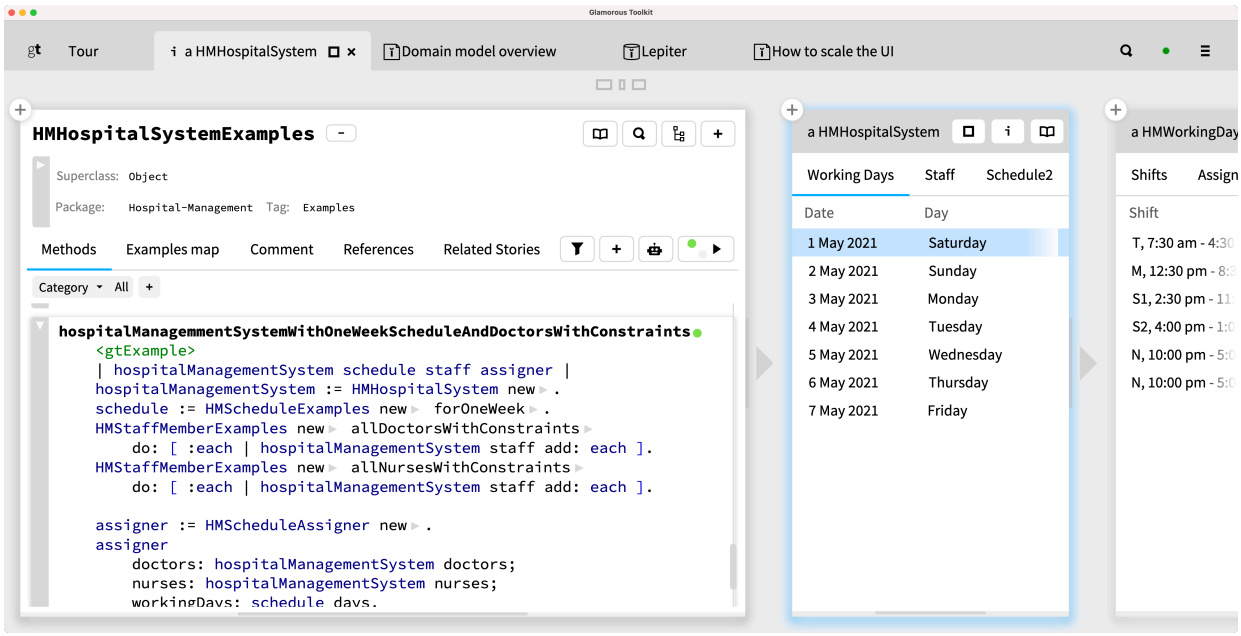


Fig. 2: A sample in-IDE executable scenario

explores how a system behaves under different conditions. Note that such scenarios can be fully generated from an in-IDE user interface [19].

## III. Live documentation

Various project-related documents in our approach are first-class entities themselves. One can dynamically create several documents that consume different artifacts and explain specific aspects of a running system. Such a document can help explain library APIs to a developer. For non-technical stakeholders such documents can aggregate project-related information for requirements-related entities. With first-class artifacts readily available, documents can serve as interactive tutorials in addition to static specifications. In this section, let us see how we can use the discussed artifacts to compose interactive and live documentation. We will discuss two situations and a type of documentation that might help a specific stakeholder to accomplish a goal in that situation.

### A. A Kanban board

Now, let us consider a non-technical stakeholder, such as a product owner, who needs an overview of the up-to-date

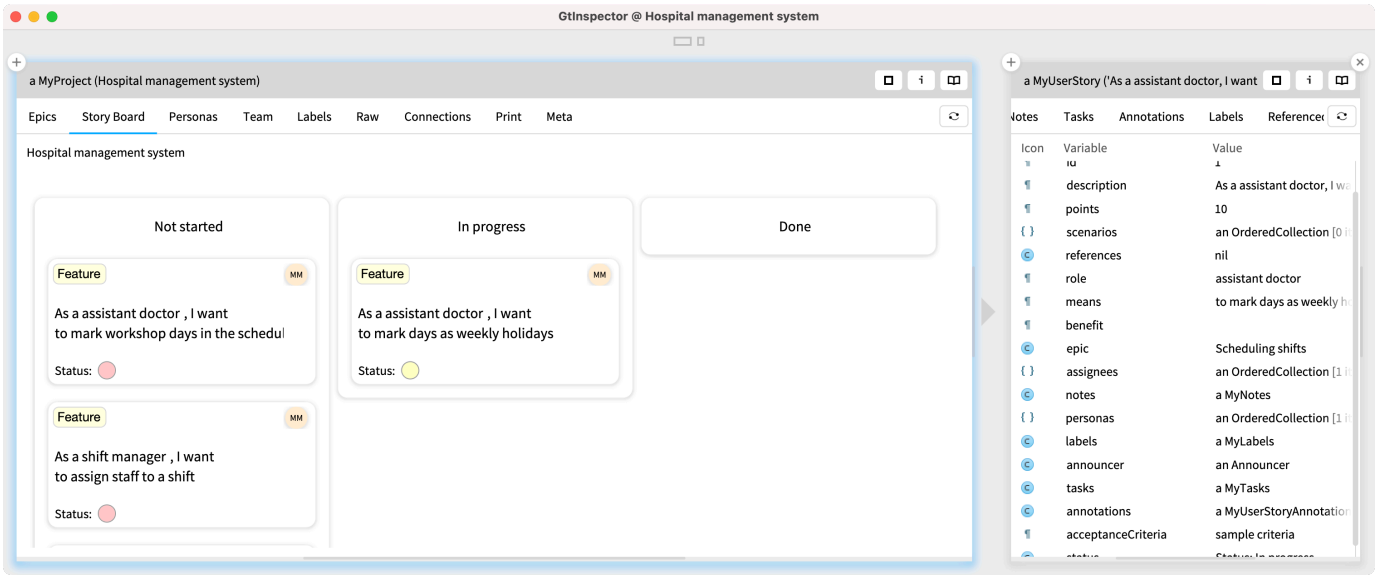


Fig. 3: A sample in-IDE live Kanban board

progress of the project. Traditionally, they 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, they will need a similar overview in an IDE itself.

A Kanban board can serve as project documentation that helps product owners track the live progress of the project. In Figure 3, 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. This representation gives a non-technical stakeholder a quick visual overview of the current progress status of a particular project. Each user story is a first-class entity and after clicking on the card, its details are accessible right next to the card representation.

### *B. An interactive tutorial*

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 4, 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 can execute scenarios inline and explore the results right next to it without losing the context.

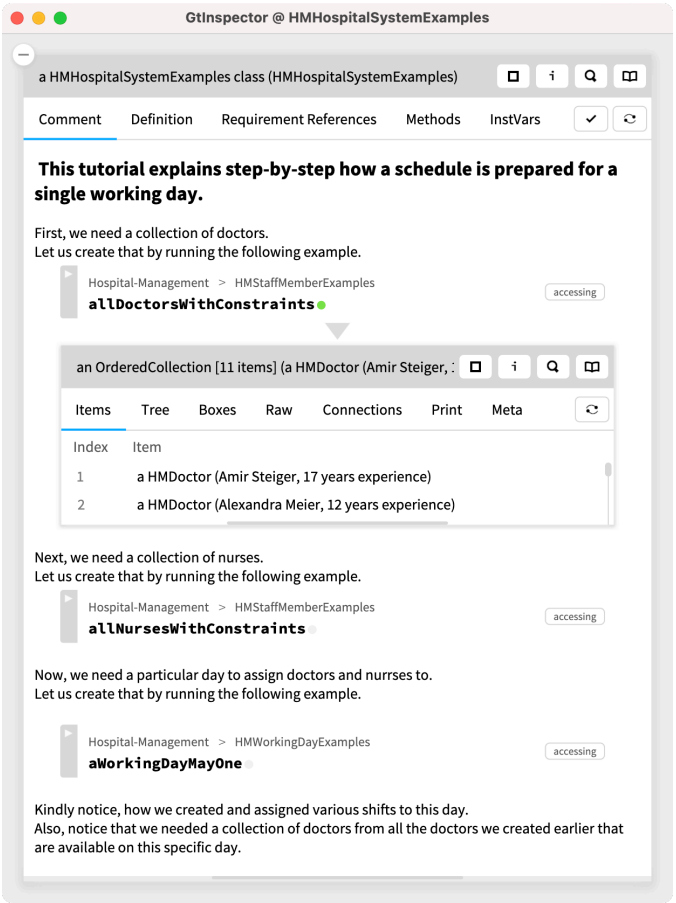


Fig. 4: Interactive in-IDE live tutorial

## IV. 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. The online 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 all the authors 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, please refer to both ".xlsx" files. The participants had varying experiences with software development and agile methodologies, ranging within 7-20 years. 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.

## V. Conclusions and future work

To avoid scattering 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 development-related artifacts in one platform, eliminating a need to recover trace links. We discussed how can we compose first-class artifacts into live documents for various types of stakeholders to accomplish a certain goal. We presented an advanced prototype implementation of three artifacts in Glamorous toolkit and a discussion of the advantages of our approach. We also conducted a semi-structured online pilot survey with practitioners and researchers to evaluate the potential of our approach for artifact management. The initial results are encouraging for us to continue with this research line and the feedback from the pilot survey participants will be taken into account before we proceed with a full user study.

As future work, we plan to extend the current implementation of our running example, among other things, providing user interfaces for non-technical users. To evaluate the actual potential of our approach, we plan to conduct a controlled experiment with a mix of practitioners and researchers. We are interested in observing how the participants interact with our example implementation, and recording values of various task-related metrics, such as time to complete one task or the number of context switches required to accomplish assigned tasks.

## Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assistance" (SNSF project no. 200020-181973, Feb. 1, 2019 - April 30, 2022). We acknowledge Aliaksei Syrel for his contribution in the implementation, and Pooja Rani, Alexandre Bergel, and Norbert Seyff for reviewing the manuscript.

## References

[1] O. Liskin, "How artifacts support and impede requirements communication," in *Requirements Engineering: Foundation for Software Quality*, S. A. Fricker and K. Schneider, Eds. Cham: Springer International Publishing, 2015, pp. 132–147.

[2] T. R. Silva, J.-L. Hak, and M. Winckler, "Testing prototypes and final user interfaces through an ontological perspective for behavior-driven development," in *Human-Centered and Error-Resilient Systems Development*. Springer, 2016, pp. 86–107.

[3] A. Garcia, T. S. da Silva, and M. Selbach Silveira, "Artifacts for agile user-centered design: a systematic mapping," *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.

[4] P. Ghazi and M. Glinz, "Challenges of working with artifacts in requirements engineering and software engineering," *Requirements Engineering*, vol. 22, no. 3, pp. 359–385, 2017.

[5] R. Damaševičius, "Analysis of software design artifacts for socio-technical aspects," *INFOCOMP Journal of Computer Science*, vol. 6, no. 4, pp. 7–16, 2007.

[6] L. Carvajal, A. M. Moreno, M.-I. Sanchez-Segura, and A. Seffah, "Usability through software design," *IEEE Transactions on Software Engineering*, vol. 39, no. 11, pp. 1582–1596, 2013.

[7] O. Gotel and A. Finkelstein, "Contribution structures [requirements artifacts]," in *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*. IEEE, 1995, pp. 100–107.

[8] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[9] O. C. Gotel and C. Finkelstein, "An analysis of the requirements traceability problem," in *Proceedings of IEEE International Conference on Requirements Engineering*. IEEE, 1994, pp. 94–101.

[10] C. J. Stettina and E. 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*. IEEE, 2013, pp. 1–12.

[11] E. Aghajani, C. Nagy, O. L. Vega-Márquez, M. Linares-Vásquez, L. Moreno, G. Bavota, and M. Lanza, "Software documentation issues unveiled," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1199–1210.

[12] M. Glinz, "Should requirements be objects?" in *Tutorial Position Paper, 14th Annual International Symposium on Systems Engineering*. Citeseer, 2004.

[13] H. Kaindl, "The missing link in requirements engineering," *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 2, pp. 30–39, 1993.

[14] "Glamorous toolkit," http://gtoolkit.com/, accessed: 2021-04-03.

[15] A. Chiş, O. Nierstrasz, A. Syrel, and T. Gîrba, "The moldable inspector," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2015, pp. 44–60.

[16] G. Lucassen, F. Dalpiaz, J. M. E. van der Werf, and S. Brinkkemper, "The use and effectiveness of user stories in practice," in *International working conference on requirements engineering: Foundation for software quality*. Springer, 2016, pp. 205–222.

[17] I. Mahmud and V. 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)*. IEEE, 2011, pp. 157–162.

[18] M. Wynne, A. Hellesoy, and S. Tooke, *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

[19] N. Patkar, A. Chis, N. Stulova, and O. Nierstrasz, "Interactive behavior-driven development: a low-code perspective," 2021.