UNIVERSITÄT
BERN

Increasing stakeholder engagement with object cards

Bachelor Thesis

Artthik Sellathurai

from

Bern BE, Switzerland

Faculty of Science
University of Bern

28. March 2022

Prof. Dr. Oscar Nierstrasz

Nitish Patkar & Dr. Nataliia Stulova

Software Composition Group

Institute of Computer Science
University of Bern, Switzerland

Abstract

There exist several tools that enable domain experts to model a problem domain graphically. In this thesis, we investigated a selection of state-of-the-art graphical modeling tools to study their characteristics and to uncover their shortcomings regarding their support to engage multiple stakeholders.

We observed that the existing graphical modeling solutions: (1) allow users to graphically specify domain entities and relationships between them, and (2) facilitate automatic code generation from the visual specifications. We observed limited support in selecting domain-specific graphical notations, and the code generation facilities offered by the existing tools are uni-directional only, *i.e.*, from specification to code. Furthermore, we observed that the existing tools might be hard to master for non-technical stakeholders, as these tools require the user to have in-depth knowledge about the offered functionality.

To tackle these limitations, we propose an approach to iteratively create domain models inside an integrated development environment (IDE). With our approach, non-technical stakeholders can graphically create actors of any domain as first-class citizens and iteratively give them behavior. Developers create custom graphical notations suitable for non-technical stakeholders.

We demonstrate a prototype implementation that demonstrates our approach in the Glamorous Toolkit IDE with a case study.

Contents

1 Introduction

Agile development methods are used in software development to adapt software in an environment of rapidly changing requirements and have functioning prototypes along the way [19]. Modeling a domain in an agile way means incrementally creating domain models that are simple, easy to modify, and facilitate rapid feedback [2, 22]. Therefore, agile methods demand modeling that is iterative and interactive, and preferably visual [2, 33]. Apart from being tools for communication, documentation, and code generation, models in an agile context also provide a means for knowledge representation [22, 24]. Visual modeling is used to model various aspects of software [2, 39], and the visual modeling process results in artifacts like diagrams, or different visual representations of the modeled software [23]. These artifacts used in visual modeling tools help engage stakeholders of varying technical knowledge — domain experts, business stakeholders, executives, and technical teams — to take part in the iterative interactive development process [39]. Similarly, visual domain models assist project stakeholders with distinct tasks, for example, they allow domain experts to specify the problem with domain-specific notations [31, 41]. Due to different kinds of stakeholders participating in designing models, the resulting models can hold information that need not be relevant to all stakeholders [40, 41]. For instance, team members with non-technical responsibilities do not consider the detailed technical perspective on entities for their work [31], whereas developers depend on detailed technical perspectives to define and implement the functionality and behavior of entities of the domain model [31]. These contrasting perspectives on models prevent different kinds of stakeholders from discussing the requirements and the features of the software [41]. Frameworks, technologies, or

in-depth knowledge of the technical part of software programming are not needed to discuss models [39]. However, they assist technical stakeholders, such as product owners to track concrete requirements [11]. Visual modeling tools in agile development help communicate the domains, models, and behavior of the designed software to the development team in a simple and interactive way. In that development team, they help to specify the technical implementations that are needed for the software.

Various visual modeling approaches, techniques, and corresponding tools are at the teams' disposal. For instance, numerous tools use the reasonably well-defined industry-standard unified modeling language (UML) to create and communicate visual software models to different groups of stakeholders in software development [39]. Standardized UML diagrams can be used to describe the behavior and relationships of the domains entities. Often, such UML diagrams are created at the beginning of software projects and remain static in the ongoing development process [20]. In agile development, where software is developed iteratively, such un-modified diagrams may not represent the most current version of software [20]. Apart from UML, several other visual modeling approaches have been developed for different purposes. Business process modeling notation (BPMN), for example, helps to visualize and identify business entities of a business-specific process. Due to the domain-specific nature of the various BPMN frameworks, it becomes partly unusable for software development in general [12]. Apart from these standardized modeling approaches, 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 and several DSMLs [4–6, 17, 32, 43]. As we analyze these tools we see that some of them are partially unidirectional, *i.e.*, they transform visual models to code structure, or the code structure generates a visual model. Some create source code scaffolding that needs further implementation of the visually modeled entities and domains. Other tools, such as Eclipse UML Generators can create visual models from the code that has been implemented by the software developers. There are often only limited ways of having fully functioning software prototypes after the visual model has been altered or expanded. This stands in direct conflict with having an iterative and agile development process.

One concept that helped us to design our approach is that of Low-code platforms. Low-code platforms allow an organization to build software with minimal coding (mainly drag and drop) and focus more on business functionality [3]. They open the door to the visual development of the user interface and business logic but also to integration with data stores and services [1]. Another idea that we build upon is the Naked Object framework. The Naked Object framework automatically generates a user interface that supports creating, reading, updating, and deleting (CRUD) operations for domain objects to bootstrap a business application [35]. The Naked Object framework emphasizes the creation of domain objects with behavioral completeness to avoid tangling the business logic to several objects [37]. Our principle idea to tackle the aforementioned limitations is that the *model is code* and should be developed incrementally, iteratively, interactively, and preferably in an IDE with helpful visual representations [24, 33, 45]. With our approach, the source code is the domain model in text form, and the domain model is the visual representation of the code. This idea and this view of models and code facilitate the concept of changing stakeholder contexts

within an IDE. The goal is that, given appropriate interfaces for interaction within an IDE, domain experts without any technical or programming background can be included in the agile development process. Furthermore, as enterprise knowledge changes, the domain models change. To solve this, our approach enables non-technical stakeholders to visually, incrementally, and iteratively model domain entities with all their relationships and behaviors. Additionally, our prototype tool creates the code backbone of these created models and entities. The possibility to develop in a fully reflective IDE, such as the Glamorous Toolkit IDE,[1] has enabled us to create a bidirectional tool that reflects both the graphical model and the software code. In this thesis, we answer the following research question:

- RQ: What are the advantages of a visual modeling tool that offers multiple perspectives for different stakeholder responsibilities on a domain?
  We discuss our approach and implementation that allows stakeholders to visually model software within our tool.

To exemplify our tool we use a running example of an automated teller machine (ATM). In our ATM model, we created objects for all the ATM entities and implemented the behavior for these objects. Our approach intertwines the visual model of the ATM and its interfaces with the code, such that each represents the ATM and bank domain at any given time. The visual domain model for the ATM and the bank entities can be graphically created and extended by non-technical stakeholders, while always having a runnable code. Furthermore, our tool allows technical users, such as software developers, to implement detailed entity behavior for the ATM entities that are simultaneously reflected in the visual domain model. We used the live programming and reflective properties of the Glamorous Toolkit IDE to develop our modeling tool. With inspection possibilities (that are provided by the Glamorous Toolkit IDE) of the modeled entities in running prototypes the designed requirements of the domain model can be tested with use cases. Additional code implementation to program the behavior of the domain entities result in usable prototypes at the end of each iteration. This ensures that our tool can be used in agile development.

The thesis is structured as follows: In chapter 2, we will take a closer look at related work and some of the state-of-the-art visual modeling tools. Furthermore, we discuss their characteristics and shortcomings with a modeling example. Next, chapter 3, we present our approach and building blocks to an incremental, iterative visual modeling tool in an IDE. In chapter 4, we conclude our thesis and discuss possible future work in this field of research and with our approach. In the addendum of the thesis, the readers will find a thorough introduction to our tool and a tutorial following a simple example.

[1] "The Moldable Development Environment", accessed 4th December 2021, `https://gtoolkit.com/`

2

State of the Art

In this chapter, we will take a closer look at the related work and discuss a representative modeling tool and its specific characteristics to support domain modeling. Several studies have discussed how to improve software development with visual modeling tools. In this work, we only refer to approaches that focus exclusively on agile software development.

Laurenzi *et al.* argue that the main purpose of domain models is to capture enterprise knowledge [24]. However, they specify that due to the frequent change in enterprise knowledge, models have to be redesigned continuously as they become outdated. Various authors have discussed the benefits and limitations of visual domain modeling tools. For example, Naujokat *et al.* suggest that graphical modeling is desirable and efficient [33]. They found that IDEs are mostly used for language workbenches and environments for meta-modeling, and building domain-specific tools is a complex, repetitive, and tedious task. They suggest applying the concept of domain specialization also to the (meta-)domain of "domain-specific modeling tools." Zweihoff *et al.* argue that graphical domain-specific languages have turned out to be particularly suitable for domain experts without any programming background [45]. They point out that the enormous effort required to develop domain-specific graphical modeling tools leads the industry to avoid building custom graphical modeling tools. Ambler *et al.* describe principles for modeling in an extreme programming environment [2]. They suggest embracing an incremental change in the model and to use just enough distinct models to enable rapid feedback as a key to agile development. Finally, Vaupel *et al.* present a process to build a domain-specific IDE, an environment specifically built to develop for a

certain domain, that includes model editors and code generators [42]. They describe how domain-specific IDEs are an enabling technology for the model-driven development of specific applications.

2.1 Relevant modeling approaches

Apart from studies that argue directly in favor of visual modeling tools, several other research areas with SE have contributed to advancing multiple stakeholder involvement in domain modeling.

UML. Due to the standardized concepts, the UML is widely used in software development [7, 11, 18, 39]. Because most stakeholders understand the UML diagrams, software development teams can use UML to communicate requirements with any stakeholders [11, 18, 39]. Recent work includes creating visual DSMLs based on UML. These approaches rely on the advantage that UML is a generally accepted modeling language also known by the non-technical stakeholder [14, 16]. But although UML can be viewed as the 'de facto' standard, it is not universally adopted [**?** **?**]

BPMN. The BPMN visual model represents process entities primitively, making it suitable for any non-technical domain experts to understand and expand [12, 21, 27]. Mostly used in analyzing business processes, BPMN helps to create new business processes, find bottlenecks in the existing ones, and streamline already-existing processes [12]. The use of BPMN in the Internet of Things (IoT) has extended its application areas [27]. Another approach was proposed by Ivanchikj *et al.* to generate a BPMN out of the notes about the requirements taken in an interview [21].

DSML. As the name suggests, Domain-Specific Modeling Languages (DSLM) are used to model objects in a form that is specific to a certain domain. DSMLs are recently used to widen the boundaries in which domain experts can test or design applications, several publications try this approach with specializing the building blocks [25, 30, 38, 44]. Vještica et al. provide an application of a DSML, which is capable of modeling production processes that are ready for automatic code generation. Sousa et al. argue that each time a DSML approach has to be realized in a different domain, substantial re-implementation has to take place and present their work towards a generalization of the re-implementation process.

Low-code platforms. Low-code platforms improve the software development process by increasing stakeholder involvement and decreasing the technical expertise to use development environments [3]. Low-code platforms provide a development environment to create software through a graphical user interface instead of textual programming. The Low-code platforms rely on model-driven design, automatic code generation, and visual programming. Important characteristics are model-driven development, reusability through templates, widgets, and plug-ins, support throughout the software cycle, and optional cloud or on-premise hosting options [**?**]. The approach of Low-code platforms is used to ease the access of developer environments to non-technical domain experts and professionals [28, 29]. This goal is achieved by having a model-driven development environment with a graphical user interface that allows non-technical stakeholders to visually program. This, in turn, creates software, by means of automatic

code generation. Metrôlho et al. anticipate an increase in the demand for professionals with skills in the areas of ICT and present several case studies and describe a strategy, where a Low-code platform is used to reskill STEM professionals, to perform ICT professional activities.

Naked object framework. The naked object framework is interesting to agile software development because the users can focus on the creation of business classes and do not need to concern themselves with the implementation of any other layers or interfaces [15]. The framework can be used to create business applications in model-driven development environments [15, 37]. Principles of the pattern behind the framework contain that (1) all business logic should be encapsulated onto domain objects, (2) the user interfaces should be direct representations of the domain object, and these (3) user interfaces shall be automatically created from the definitions of the domain objects. The naked object framework then automatically generates a user interface that supports CRUD operations for domain objects to bootstrap a business application [35]. To avoid tangling the business logic to several objects the framework emphasizes the creation of domain objects with behavioral completeness [37]. The naked objects pattern was once used by the Department of Social Protection in Ireland to replace an existing system for the administration of child benefits. Since then, the pattern was redeveloped around two naked object frameworks which now form the basis for future development.

In the following section, let us take a closer look at one of the representative visual modeling tools, *i.e.*, GEMOC, and discuss its features and the supported modeling workflow.

2.2 GEMOC

Various studies in various research areas in software engineering have proposed modeling tools. It is a tremendous amount of work and out of scope for this thesis to analyze all the existing graphical modeling tools. Instead, as a representative, we picked one, *i.e.*, GEMOC,[1] which also boosts modeling in an IDE and seems to be well-cited in academia, to analyze their support for domain modeling. Next, let us see how GEMOC enables graphical domain modeling and its limitations for agile and collaborative modeling.

GEMOC Studio offers two workbenches needed for different tasks, one being the language workbench, which is used by language designers to build DSMLs. The modeling workbench is used to model domains with the modeling languages that have been created in the language workbench. These workbenches are used by distinct stakeholders — the language workbench is a tool for the technical language designer to create a language, while the modeling workbench on the other hand is a tool for the non-technical model designer. A classical implementation in GEMOC would need several steps as shown in Figure 2.1: (1) create a language, (2) make an executable language, (3) define the runtime environment for the language, (4) compose executable languages, (5) create models, and finally (6) execute, debug and animate those models.

Before using the modeling workbench, language designers and technical stakeholders need to put

[1] "The GEMOC Initiative", accessed 4th December 2021, `http://gemoc.org/`

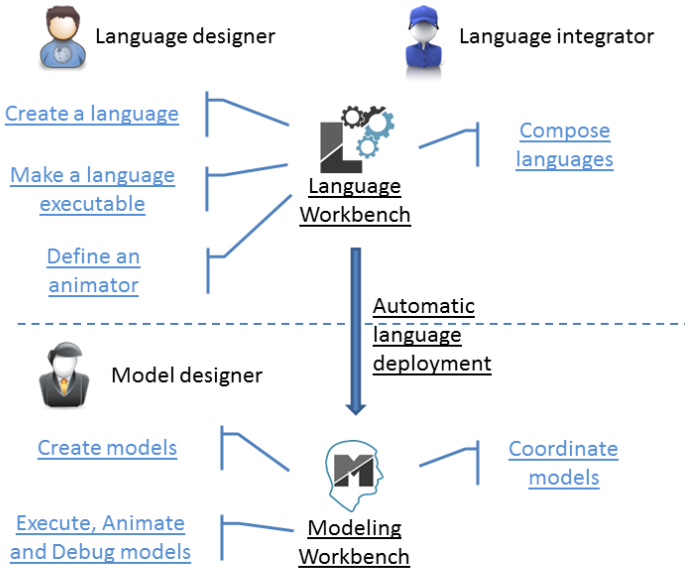


Figure 2.1: The process steps suggested by GEMOC

an effort into defining a business domain. As a consequence, a modeling language— specific to the domain— has to be designed by language designers. Later, when modeling domains with non-technical stakeholders, if the domain has changed or expanded, then the modeling language needs to be updated. Although the language can be improved incrementally, the use of different tools serially by different stakeholders rejects the agile manner. Due to the division of medium (*i.e.*, not the same tool) between the language and the modeling tool, the length of the development iterations might increase, leading to slower development. It leads to even bigger issues when an error in the modeling language is identified once the code generation and implementation begin. One needs to reverse multiple layers and repeat the modeling steps. In conclusion, the process of creating a domain model with GEMOC relies heavily on a waterfall-like development process and seems unfit for agile development.

2.3 Perceived limitations of the discussed modeling approaches

Although various approaches try to close the gaps between non-technical and technical stakeholders, not all of them can be used in agile development easily. Most of the approaches require initial work before modeling can begin. Below we summarize the perceived limitations of the existing approaches and tools:

- *Highly specific groundwork and adjustments.* Existing modeling tools require preparation steps that can only be done by language experts. DSMLs that are defined in volatile or niche domains need to constantly evolve to fit their revised domains.
- *Model and code are not the same.* Code does not reflect existing visualizations of the domain model and vice versa. In GEMOC, the model layer and the source code layer are separated. Although the visual model changes the code backbone of the model, simultaneous changes in the code cannot be represented in the visual model. The same applies to executable prototypes that can only be created after several necessary steps of modeling and implementing code.
- *Long feedback loops.* To further alter visual representations of run-time objects in the model, the modeling process needs to be halted and the framework for the visualizations needs to be expanded or changed. This approach requires highly trained professionals for many critical steps in the process. Likewise, not being able to work on the model or the code in certain phases of development rejects the idea of agile development and increases the need for communication and coordination.

All of the approaches have some shortcomings and none of them unifies the agile development process for technical and non-technical stakeholders alike. With our approach, we suggest that the path from an idea/requirement of a domain to a specific domain model can be shorter, and all stakeholders can work on it throughout the software development process in an agile manner. By using a tool with multiple specific visualizations and representations of the domain model that have been tailored to the needs of the stakeholders, every stakeholder shall be enabled to work on the domain model iteratively and incrementally.

3

An Agile modeling tool

In this chapter, we will start by looking at the key principles of our approach and the background to justify our principles. Afterward, we will introduce the building blocks of our tool *Object Cards* and describe our running ATM example. Lastly, we discuss and analyze the implications of our approach with its key principles and implementation attempt.

To address the limitations described at the end of chapter 2, one of the key principles of our approach is that *all stakeholders can build models interactively*. To achieve this we wanted Object Cards to be built in a visual playground where graphical elements can be dragged, dropped, and further manipulated, especially by non-technical stakeholders. In this visual playground, users will be supported to create the domain model with a graphical user interface that can be tailored to their needs. For instance, one of the graphical representations of the domain model will allow users to describe the purpose of domain entities, or another one that can test implemented use cases by executing a command and step through the responsible domain entities. Through different representations of the same domain model, distinct stakeholder groups can engage continuously in the development process. Another key principle of the approach consists of the idea that *behavior can be added incrementally and iteratively*. Object Cards shall enable the users to add to the domain model, test the changes and alter the existing domain model. In a visual representation, which consists of a source code editor, software developers can add behavior logic to domain entities as usual. In addition, non-technical stakeholders can also add, describe and manipulate domain entities and their behavior. To further this principle Object Cards will also use an execution framework that allows users to

test use cases. As we will see later, a lot of the foundation for the implementation of these principles was laid by the Glamorous Toolkit.

3.1 Background

Domain models and software implementations are viewed as distinct artifacts in the development process [45]. This divide is also visible in the programming infrastructures where the source code is only represented as text and the IDE isn't designed to represent domain concepts in domain-specific ways, but instead, represents all objects in a generic format [34]. The gap between models and code can impair communication between stakeholders because domain experts might not be able to productively navigate an IDE [45]. Without having a contextual and understandable representation of the work that has been implemented in a software project, different stakeholder groups have no common denominator that can be used as a basis for discussing domain models. Domain experts without any technical or programming knowledge must wait for a prototype to determine if the domain models are implemented correctly. As a result, the interactivity of certain stakeholder groups is highly limited. This can lead to longer feedback loops which stand in contrast to wanting to have rapid feedback in agile development for incremental and iterative change [2].

Different models and visualizations are needed for distinct tasks that different stakeholders are responsible for [31]. In practice, the creation and maintenance of these multiple visualizations form a bottleneck [24, 45]. The bottleneck is the enormous effort to develop required domain-specific graphical modeling tools [45]. With the Glamorous Toolkit IDE, the concept of *moldable tools* helps us to create visualizations that purposefully represent domain-specific information to all stakeholders [8]. Moldable tools help to create moldable representations and visualizations, that can be altered and new ones can be added. These moldable tools are the foundation for all the different kinds of representations and visualizations that can be developed for the IDE.

3.2 Building blocks

First, we will take a look at the core components and tools that are at our disposal by Glamorous Toolkit. In the last part of this section, we then explain the main building blocks of Object Cards: *widgets* and *object cards*.

The *Playground* provides us with an entry point to Object Cards. This live coding notebook enables us to start our tool with an empty domain model or open running examples with test cases. In these Playground windows (see Figure 3.1) users shall be able to build their domain model.

The *Inspector* (see Figure 3.2) lets us define custom views for every object. We can leverage the default representations for text, numbers or any other graphical symbols.

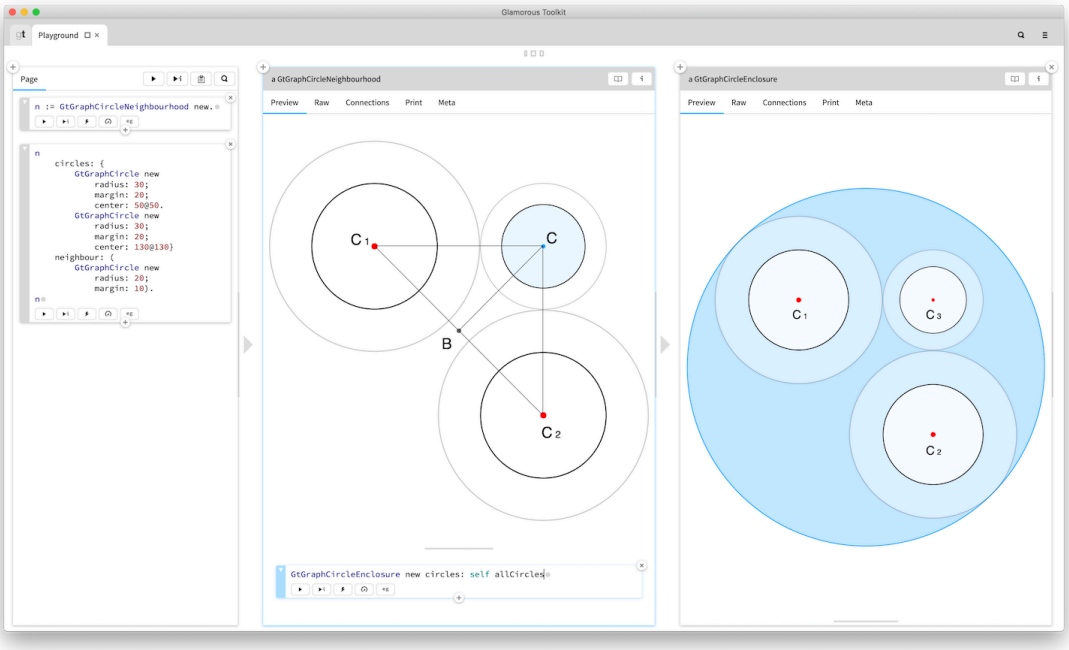


Figure 3.1: Playground embedded in an Inspector.

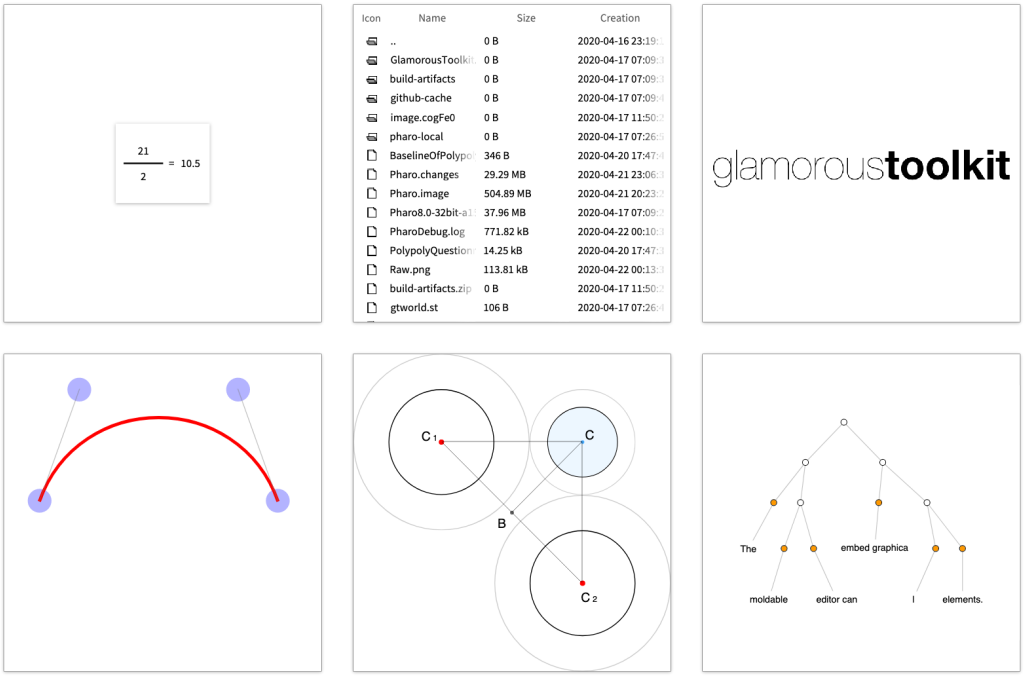


Figure 3.2: Different visual representation for different objects

The *Bloc graphical stack* (see Figure 3.3) is the foundation for the Glamorous Toolkit. Additionally to the components, we mentioned we use graphical representations of arrows, lines, and buttons to visualize and represent the domain model.

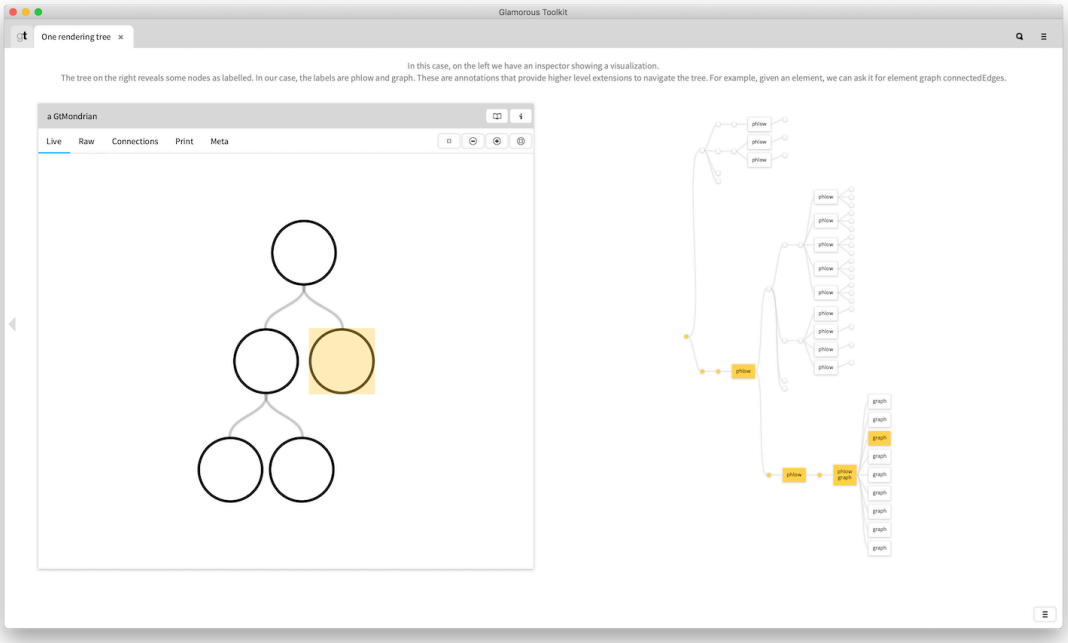


Figure 3.3: The Inspector pane on the left with a visualization using the bloc graphical stack

3.2.1 Widgets

Most IDEs that support debugging have object inspectors that allow developers to inspect run-time objects. These object inspectors often have standard representations which show attributes of a run-time object and their values [9]. These representations are primarily useful for developers and technical stakeholders. On the other hand, non-technical stakeholders need to see domain objects with domain-specific representations. In end-user software products, such as web applications, graphical user interfaces (GUIs) are used to display objects with an understandable representation for the end-user. For instance, a table with numerical values is more useful to end-users than a raw JSON standard object inspector representation of a table.

A *widget* is a domain-specific representation of a run-time object. It is essentially a piece of code that generates a domain-specific visual representation of an object. This visual representation uses elements from the Bloc graphical stack, such as text boxes, arrows, or buttons, to visualize characteristics of domain entities in an object-specific way. As an example, the piece of source code shown in Listing 1 generates a domain-specific representation for a textual description field of a domain object with the help of `BrEditor`, an element from the Bloc graphical stack.

```
commentEditor
    ↑ BrEditor new
        look: BrEditorLook;
        padding: (BlInsets all: 5);
        hMatchParent;
        vExact: 30;
        text: '...';
        yourself
```

Listing 1: Sample source code for a widget

It is the developer's responsibility to create and maintain widgets that are useful for any stakeholders and subsets of stakeholders. The goal is that any aspect or perspective of a domain object can be displayed by widgets.

Another example of a widget is the *Coder* widget, where developers can implement logic and behavior, similar to classical text-based IDEs (see Figure 3.4). Here we leverage the `GtMethodCoder` component from Glamorous Toolkit, to create a widget with object methods that can be edited. Furthermore, due to the capability of the `GtMethodCoder` this widget can be used to execute code snippets to test use cases.

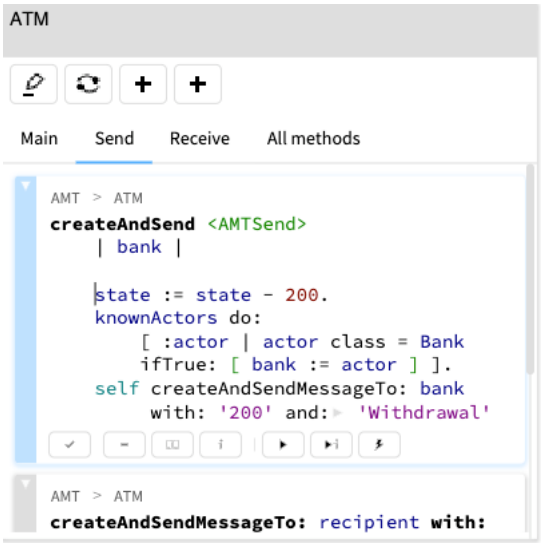


Figure 3.4: The *Send* tab with the *Coder* widget enables us to edit, execute and visualize code in the domain model with a variety of execution commands in the bottom of the code snippet box.

In summary, Object Cards leverages the Glamorous Toolkit and Bloc graphical stack components to create and build widgets for any characteristics of a domain entity. We suggest that the key principle of our approach that stakeholders can build models interactively is encouraged by Object Cards with the possibilities of widgets. Widgets enable users to see and interact with domain object characteristics in an object-specific way they can understand. These widgets can be used to close the gap between models and

code mentioned in section 3.1. Domain experts without any technical or programming knowledge mustn't further wait for the finished prototype to determine the correctness of the domain model. With widgets there is no need to know how to navigate an IDE and understand an object inspector, any stakeholder can have object- and stakeholder-specific representations of the domain model. Finally, non-technical domain experts can further engage in the development process.

3.2.2 Object cards

There are not many IDEs that enable users to see multiple domain object representations together in context. To observe different aspects of run-time objects developers would need to navigate through the options of the object inspector's user interface. As mentioned in subsection 3.2.1, these representations are primarily made for developers that know how to navigate an object inspector. For example, in modern smartphone or smartwatch operating systems end users can customize their displays to show relevant information and grant access to the most used features. Object Cards tries to enable any stakeholder to examine multiple customized contextualized representations of the domain object characteristics they need. To live up to different stakeholder-specific representation needs Object Cards uses `BrTab` from the Bloc graphical stack, to deliver multiple different tabs with different widgets. Simply put, object cards use tabs, these tabs use widgets. Each object card can have multiple tabs, these tabs are used for a stakeholder-specific concatenation of widgets. For example, one object card can have multiple tabs, with each tab for each stakeholder group. These tabs can contain more than one domain entity characteristic, so a condensed representation can show any stakeholder all information they need.

An *object card* consists of *tabs* of widgets and is a tailored display of multiple widgets of the same domain object. Tabs represent specific domain object representations tailored to the stakeholders' needs. An object card can have more than one tab and one tab can have more than one widget and widgets can be reused for other tabs. To realize object cards in Object Cards we make use of widgets and `BrTab` from the Bloc graphical stack to concatenate and display widgets. The following source code shown in Listing 2 generates a tab with a description field and an inspection field for instance variables.

```
1  mainTab
2      | canvas |
3      commentLabel := self commentLabel.
4      commentEditor := self commentEditor.
5      saveCommentButton := self saveCommentButton.
6      attributesLabel := self attributesLabel.
7      attributesElement := self attributesElement.
8      canvas := BlElement new
9          layout: BlLinearLayout vertical;
10         background: Color white;
11         constraintsDo: [ :c |
12             c horizontal matchParent.
13             c vertical matchParent ];
14         look: BrShadowLook;
15         addChild: commentLabel;
```

```
        addChild: commentEditor;
        addChild: saveCommentButton;
        addChild: attributesLabel;
        addChild: attributesElement.
    ↑ BrTab new
        look: BrGlamorousTabLook new;
        label: 'Main';
        stencil: [ canvas ]
```

Listing 2: Sample source code for a tab

These tabs are collected in a `TabGroup` also from the Bloc graphical stack. Additional to these tab groups our object cards contain general stakeholder-unspecific action buttons and a label for the name of the domain object. These are realized by different Bloc graphical stack components, such as buttons and text boxes. The following code (Listing 3) shows Object Cards of object cards.

```
initialize
    super initialize.
    self layout: BlLinearLayout vertical.
    self background: Color white.
    self look: BrShadowLook.
    self
        constraintsDo: [ :c |
            c horizontal exact: 400.
            c vertical exact: 400 ].
    nameLabel := self nameLabel.
    saveNameButton := self saveNameButton.
    refreshButton := self refreshButton.
    addActorButton := self addActorButton.
    linkActorButton := self linkActorButton.
    tabGroup := self tabGroup.
    buttons := BlElement new.
    buttons layout: BlLinearLayout horizontal.
    buttons
        constraintsDo: [ :c |
            c horizontal matchParent.
            c vertical exact: 40 ].
    buttons
        addChild: saveNameButton;
        addChild: refreshButton;
        addChild: addActorButton;
        addChild: linkActorButton.
    self
        addChild: nameLabel;
        addChild: buttons;
        addChild: tabGroup.
    ↑ self
```

Listing 3: Sample source code for an object card

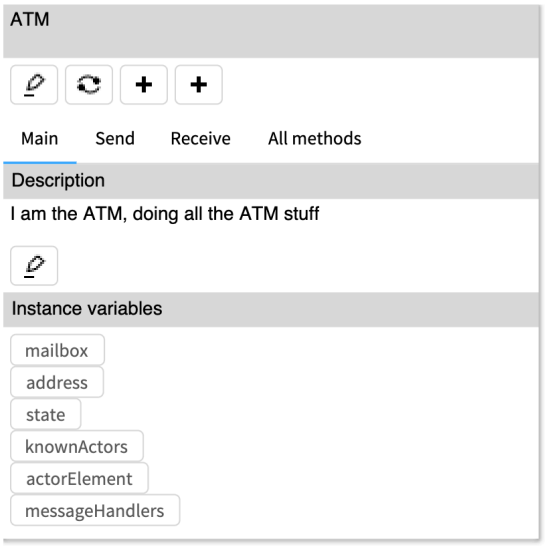


Figure 3.5: An ATM object card on the All methods tab with widgets for source code editing and use case testing of the ATM domain object. Under the ATM label one can find the action buttons not specific to any stakeholder group.

Figure 3.5 shows the object card of the `ATM` object. This object card has multiple tabs, `Main`, `Send`, `Receive` and `All methods`. The main tab shows us a textual description of the domain entity and a selection of domain-relevant instance variables of the ATM object instance. In Figure 3.4 we can see another tab with another stakeholder-specific representation for source code editing and use case testing.

In summary, we leverage great parts of the Bloc graphical stack, to gather stakeholder-needed widgets in tabs. These tabs are then gathered in an object card, that also has general action possibilities that are stakeholder-unspecific. These actions can contain multiple commands, and Object Cards mainly uses these actions to connect related domain entities or add new related domain entities.

We suggest that the second key principle of our approach that behavior can be added incrementally and iteratively is possible in Object Cards with object cards. Object cards can gather all needed representation for any stakeholders or stakeholder groups. Any stakeholder or stakeholder group can have multiple tabs for specific needs and perspectives on a domain entity. All these different perspectives on the domain entity might provide a common denominator that can be used as a basis for discussing the domain model. As described in section 3.1 this enables the possibility of shorter feedback loops and rapid feedback, which in return facilitates the foundation for incremental and iterative change to the domain model.

3.3 Running example

Automatic teller machines (ATMs) and all their associated actions one can perform with them have served as a well-known example to motivate better design decisions [10]. ATMs operate outside of the

opening hours of a bank and therefore are widely used for several account-related transactions, such as cash withdrawals and money transfers. ATMs across the globe behave similarly to exhibit user-friendly workflows. Often, users can withdraw a fixed amount that the ATM suggests or provide valid user input. Once the user selects and confirms the desired amount, the ATM communicates with the bank interface to check whether the user has a sufficient account balance. If the user has a sufficient balance, the bank debits the amount and signals the ATM to return cash to the user. The ATM itself changes its state due to the dispensing of cash that happens in the machine.

With most ATMs, when trying to withdraw an amount that exceeds the daily limit, the transaction is aborted after the user selects a certain amount, and the machine returns the bank card to the user. The only feedback the user gets is a message that declares the daily limit has been exceeded, without any additional information about the actual amount that exceeded the daily limit. In case the user still wishes to withdraw the cash, she needs to retry with a different amount without any guarantee of success. Although it could be argued that this is a security feature against prying eyes, the additional steps needed to check the account balance could be decreased when such a use case occurs. A more end-user-friendly design could be that the ATM has an option to show the account balance information after the feedback that the daily limit has been exceeded. This feature still demands an action from the user but decreases the needed steps to check the account balance in case of an exceeded daily limit. It could show a warning about exceeding the daily limit when the user selects the desired amount, or better yet, correct the desired amount to a possible amount with an info dialog that needs to be accepted by the user. With the last option, the actions needed to find an acceptable withdrawal amount are highly decreased and the user still needs to accept the transaction.

The ATM example is interesting as it involves a couple of actors that communicate with each other, update their states upon executing specific functions, and the visual feedback they provide to the user improves the overall user experience. One should model multiple scenarios to test and see if the domain model reflects a better design that satisfies user goals and the overall experience.

3.4 Our proposed workflow

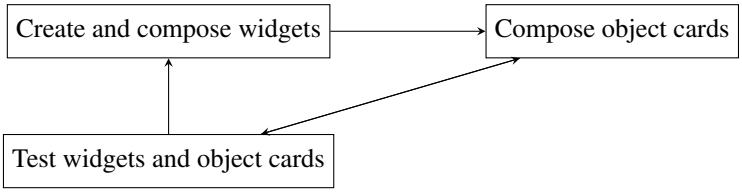
To get started with the implementation of our building blocks all stakeholders need to identify and express their representation and visualization needs. A software developer might need the classical source code representation of a domain entity. On the other hand, a non-technical domain expert might need to observe and test instance variables. Another stakeholder might need to see the domain entities in a bigger picture with descriptions, constraints, and existing connections to other domain entities. For all these needs existing widgets or object cards can be reused, but for new visualization requirements or initial work with Object Cards, new widgets or object cards might be needed.

Let us consider that we have three stakeholder groups, that have the same visualization needs within each group. Our project group is tasked with the challenge to build our described ATM scenario, with the last use case, as seen in section 3.3. Group A is a group of non-technical domain experts, group B consists

of software developers, and lastly, group C is made up of stakeholders that test the domain model. In one of these three groups, we have Joseph, a developer who is part of group B.

Group A is mainly interested in the account balances and the roles and responsibilities of each domain entity. Group B is interested in the behavior and logic of the domain entities, and group C is testing use cases.

Our proposed workflow divides the process of creating the modeling environment into three steps: (1) creating and composing widgets, (2) composing object cards, and (3) testing object cards. If new visualization needs arise while using the object cards or due to domain requirements changes, we can start again to adapt, create and compose widgets. We detail below each step.



3.4.1 Create and compose widgets

In the requirements for the ATM example, group A needs to be able to see the account balance of an end-user account. Group B wants to see all behavior and logic of the domain entities. Furthermore, group C wants a visual representation where they can test use cases. To fulfill all their visualization needs Joseph then starts to perform a differential analysis between the already existing widgets and the widgets still needed. For group A he creates a `attributesElement` widget (Listing 4) that shows the instance variables of a domain object where also the account balances will be stored.

```
attributesElement
    ↑ BrSimpleList new
        padding: (BlInsets all: 5);
        hMatchParent;
        vMatchParent;
        items: {};
        stencil:
            [ :n |
            BrButton new
                look: BrGlamorousButtonWithLabelLook;
                label: n;
                action: [ :b |  b phlow spawnObject: (actor instVarNamed: n)  ];
                yourself ] yourself
```

Listing 4: Sample source code for a widget that shows the instance variables

Additionally for group A he creates a `allMethodsList` widget (Listing 5) that lists domain object specific methods to comprehend the progress of the developers and analyze the responsibilities of each domain object.

```
allMethodsList
    ↑ BrSimpleList new
        padding: (BlInsets all: 5);
        hMatchParent;
        vMatchParent;
        items: {};
        stencil:
            [ :m |
            (GtMethodCoder forMethod: m)
                object: actor;
                yourself ] yourself
```

Listing 5: Sample source code for a widget that shows the instance variables

For the software developers, he creates a widget where all methods can be analyzed and altered, similar to Listing 5. Lastly for group C, he can create a widget similar to the one created for the developers, but these widgets only contain test methods.

3.4.2 Compose object cards

After creating widgets for every representation need, Joseph can generically create tabs for the object cards with the widgets created before. For group A, which is interested in the bank account balances and the roles and responsibilities, he creates tab `Main` (Listing 2) that contains a textual description of the domain entity, its instance variable that shows `state`, where the account balance can be observed. Additionally for group A, he adds the widget with the object-specific methods. For group B, which needs to see the methods of the domain entities, he creates an entirely different tab `All methods` that lists all the methods. For group C, which wants to test the use cases, he creates a tab `Send` and `Receive` similar to the one for group B, but with the restriction that only use-case test methods are shown. Lastly, he composes the object card with every created tab `Main`, `All methods` and `Tests`.

3.4.3 Test widgets and object cards

While modeling the domain all stakeholder groups can assess their given representations of the domain entities and discuss with Joseph if their visualization needs have been met, or if Joseph needs to alter existing widgets. Furthermore, some stakeholders might need new visualizations which Joseph can incorporate by iterating through the previous steps and lastly maintaining the widgets, tabs, and object cards that are used in the project groups.

More information on how to use our tool to build a domain model, with preexisting widgets and domain cards a manual can be found in the chapter Appendix A.

3.5 Discussion

In this section, we will elaborate on how we tried to address the identified limitations and how follow-up research could validate these claims.

Existing modeling tools require preparation steps that can only be done by language experts. By using widgets and reusing object cards, preparation steps for new projects could be reduced. If new visualizations are needed, the reusability and modularity of the object cards and widgets enable the development team to rapidly adapt to those needs. The cost to create new visualizations can be validated by the effort needed by a developer to create new visualizations. This effort could be measured as the amount of additional code needed to create new representations.

Different aspects and perspectives on the domain model cannot be depicted in one tool. By having theoretically countless widgets and object cards any visualization need can be satisfied. Follow-up research could gather visualization needs from non-technical stakeholders that couldn't be realized in state-of-the-art tools and try to implement them.

Code does not reflect existing visualizations of the domain model and vice versa. Due to the possibility to have object cards represent the source code the model and its domain objects are tied together. To validate this claim one would need to either create an interface to import the source code of a software project and see if it depicts its domain model, or create an interface to import existing models and quantify how much of the behavior is reflected in the source code.

Long feedback loops. By providing all possible visualization needs for any stakeholder, and by having the model reflect any aspect of the domain, the communication overhead, and length of feedback loops between distinct stakeholder groups decrease. Further research could validate this by introducing the tool to one of two distinct software project groups with similar tasks and measuring the feedback and communication needed to finish a project.

3.6 Implications

The source code is a model.

Maxims such as *"Programming is Modeling"* give us a new approach to bring models closer to code [34]. By having an IDE that supports developed representations of domain concepts in multiple useful domain-specific ways, the feedback between the stakeholders can be better. When any stakeholder can choose their understandable representation and tools for the domain models the development process becomes more transparent. Domain experts can implement changes and additions to the domain models without having to wait for a development artifact, like a prototype. Non-technical stakeholder interaction with the domain model is increased whilst fitting into the agile development process of the software

development team.

An IDE is the main tool of developers to build a software system, so the notion to create useful interactive representations fits here [8]. Having the same IDE without context switches further closes the feedback loop as any stakeholder has any representation at their disposal to make informative decisions [9]. Having an IDE that supports distinct representations, and that can be realized by developers, opens the possibilities to create domain- and business-specific visual representations. Visual representations can also be altered and molded by developers. These representations of the domain concepts can then be created for classical textual software implementation, as also for visual modeling of the domain. With these key representations, *i.e.*, source code and visual model, being close and intertwined, the domain is visually modeled with the source code and the source code is being implemented, along with the relation to the domain concepts of the visual model.

Moldable visualization.

Business-specific visualizations (such as process entities and process activities) are useful to domain experts for validating and reviewing the implementation of the requirements [31], whereas conventional textual representations of the domain model are needed for the software development team to be efficient [45]. There might be a need for process-centered visualization which helps business stakeholders to recognize bottlenecks, whereas these might not be useful for a developer [31]. On the other hand, the source code representation is of no use to the domain expert or business stakeholder but is needed by software developers to implement the behavior of the domain model and entities [31, 45].

In our running example, we can manipulate and add visualizations in the IDE by smoothly transitioning to the source code implementation in another representation in a graphical widget. Although this takes a technical stakeholder to manage it, by having the capabilities of the moldable tools at hand in the modeling process, purposeful representations and visualizations can be created where they are needed. Furthermore, additional visualizations in these graphical widgets can be added to the same IDE context where the modeling process is taking place. Having the possibility to add and change the Moldable visualization of the model at any time maximizes stakeholder involvement as any stakeholder can have an appropriate representation of the domain at any given time.

Prototypying and incremental definition of behavior.

Prototypes are used to evaluate, validate, adjust, and refine requirements [26, 36]. In agile development, distinct stakeholders use different prototypes in different ways to convey different types of knowledge [26]. Domain models represent the problem space with all the requirements, behavior, constraints, *etc.* whereas the solution space is explorable by prototypes [13]. The requirements themselves are all then satisfied by prototypes in the development process [22]. Due to the iterative need for prototypes in agile development, the cost for conventional prototypes can be high [13]. This can bring the problem that a key artifact to evaluate the requirements and review the domain model can be costly and consume developer resources to build prototypes. Even though domain experts manipulate domain models and technical stakeholders

design and build prototypes, fast feedback loops require the distinct groups of stakeholders to be capable of navigation and analysis of both.

We suggest that the domain model shall be reflected in a prototype. We try to accomplish this by always having useful representations that can be developed like any other representation as described in the previous section. Due to the conceptual polymorphism of the domain model, a prototype is then nothing but a different representation. These prototypes advance with the model, due to the binding of the evolving model to the source code and the executable prototype. With every incremental change in the domain model, for instance, if a new use case is implemented by a developer, this use case can be tested instantly in the modeling tool.

Some specialized widgets can be used to execute implemented object behavior. As can be seen in Figure 3.4 the textual programming widget has buttons to execute defined code snippets. In the running example in the textual representation that contains the source code, we can also execute the software implementation and play out scenarios. The domain objects can be tested with execution commands, as seen in figure Figure 3.4. By having both the requirements and the implemented domain model close, fast feedback loops for agile development are enabled and embraced. Any stakeholder can constantly test and analyze implemented tests and use cases with all the visual representations that are needed for their responsibilities. The domain model, the behavior of the domain entities, and the prototype co-evolve.

Stakeholder involvement.

Project stakeholders are investing resources for the development process to have a final product that meets their requirements and needs [2]. These stakeholders want to invest their time and money in the best, most efficient way possible and the environment should enable them to do so [2]. They want involvement in the development process [2]. Due to the connection of the model to the code as explained in 3.6 developers can implement entity behavior, while domain experts can add to the domain model. Both are incrementally enriching the model and all the representation and visualization. With prototypes available at any point in the development process, any stakeholder can iteratively review, validate, refine and adjust the model. Informative decisions can be comprehended by all stakeholders and interactively implemented by all relevant stakeholder groups. By having an IDE that allows any stakeholder to augment the domain model in the IDE with distinct representations, visualizations, and tools, everyone is enabled to be involved in the agile development process.

Having multiple graphical representations of the model, possibilities to create and define domain entities, and the possibility to implement source code, stakeholder involvement is possible at any time of the development process. At all times every stakeholder has an appropriate foundation for their distinct tasks and responsibilities, this is the foundation for rapid feedback and embracement of change that is needed in agile development. Furthermore, this enables any stakeholder to maximize their investment and involvement in the iterative, incremental, and interactive way agile development is supposed to be. The textual representation of source code, multiple visualizations of the domain, and the executable prototype of our running example of the ATM suggest that stakeholders could work together iteratively

and incrementally to develop software.

4

Conclusion and future work

4.1 Conclusion

In this thesis, we discussed how state of the art modeling tools are not suitable for agile development. Current tools require a huge amount of groundwork so they can be operated efficiently. Due to media disruption, a lot of the needed steps can be done neither incrementally nor iteratively. This leads to low stakeholder investment.

We have proposed an alternative tool to create domain models. We started an implementation for our suggested approach that needs further implementation and validation to prove its advantages. *Object cards* is a visual tool that relies on top of the visualization tools of Glamorous Toolkit. Our tool can create object instances and concatenate different object characteristics into grouped visualizations. Furthermore our tool propagates primitive information and triggers actions from created object instances. Lastly, as a proof of concept primitive examples for an ATM use case have been created.

4.2 Future work and validity

The next steps to improve our approach and this implementation could be the following:

- Implement interfaces to import and export source code projects and domain models

- Get real-life visualization needs from non-technical stakeholders that work with different domain models and create representations that are non-fictional

To really prove that our approach could lead to increased stakeholder engagement with object cards further implementation and validation is needed. To test the validity of our claims we suggest to test the tool with a willing software project group and accompany them in their meetings, evaluate the visualization needs and create the needed object cards. After creation of the object cards, let the project group use the tool and measure time needed for coordination and communication. By repreating surveys in the project group we could test the involvement of non-technical stakeholders and overall approval of our tool. This process might be needed to be repeated multiple times, as approval might be low and the approach and the tool needs to be adjusted to other unexpected requirements.

Another way to test the validity of our approach would be to quantify the effort needed to realize a software project in a state of the art tool and in our tool. By statistical comparison one could then analyze if any distribution of effort is significantly high or low.

5

Acknowledgement

Throughout the writing of this thesis I received a huge deal of assistance and support.

I would like to acknowledge the tremendous help from my supervisor Nitish Patkar. The countless zoom meetings and the steady guidance have helped me throughout my thesis. I want to thank you for not giving up on me.

I would also like to thank Prof. Dr. Oscar Nierstrasz. I thank you for the wise counsel and all the chances you have given me.

This has been a long journey, gladly we have arrived.

Bibliography

[1] A. N. Alonso, J. Abreu, D. Nunes, A. Vieira, L. Santos, T. Soares, and J. Pereira. Towards a polyglot data access layer for a low-code application development platform. *arXiv preprint arXiv:2004.13495*, 2020.

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

[3] R. Arora, N. Ghosh, and T. Mondal. Sagitec software studio (s3)-a low code application development platform. In *2020 International Conference on Industry 4.0 Technology (I4Tech)*, pages 13–17. IEEE, 2020.

[4] S. Boßelmann, M. Frohme, D. Kopetzki, M. Lybecait, S. Naujokat, J. Neubauer, D. Wirkner, P. Zweihoff, and B. Steffen. Dime: a programming-less modeling environment for web applications. In *International Symposium on Leveraging Applications of Formal Methods*, pages 809–832. Springer, 2016.

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

[6] M. F. Caro, D. P. Josyula, J. A. Jiménez, C. M. Kennedy, and M. T. Cox. A domain-specific visual language for modeling metacognition in intelligent systems. *Biologically Inspired Cognitive Architectures*, 13:75–90, 2015.

[7] M. R. Chaudron. Empirical studies into UML in practice: Pitfalls and prospects. In *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*, pages 3–4. IEEE, 2017.

[8] A. Chis. *Moldable tools*. Lulu. com, 2016.

[9] A. Chiş, O. Nierstrasz, and T. Gîrba. Towards moldable development tools. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 25–26, 2015.

[10] A. Cooper et al. *The inmates are running the asylum: Why high-tech products drive us crazy and how to restore the sanity*, volume 2. Sams Indianapolis, 2004.

[11] B. Dobing and J. Parsons. How UML is used. *Communications of the ACM*, 49(5):109–113, 2006.

[12] M. Dumas and D. Pfahl. Modeling software processes using BPMN: When and when not? In *Managing Software Process Evolution*, pages 165–183. Springer, 2016.

[13] M. J. Escalona, L. García-Borgoñón, and N. Koch. Don't throw your software prototypes away. reuse them! 2021.

[14] T. Eterovic, E. Kaljic, D. Donko, A. Salihbegovic, and S. Ribic. An internet of things visual domain specific modeling language based on uml. In *2015 XXV International Conference on Information, Communication and Automation Technologies (ICAT)*, pages 1–5. IEEE, 2015.

[15] F. Freitas and P. H. M. Maia. A naked objects based framework for developing android business applications. In *ICEIS (1)*, pages 348–358, 2016.

[16] G. Giachetti, B. Marín, and O. Pastor. Using uml as a domain-specific modeling language: A proposal for automatic generation of uml profiles. In *International Conference on Advanced Information Systems Engineering*, pages 110–124. Springer, 2009.

[17] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, and C. 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.

[18] H. Gomaa. *Software modeling and design: UML, use cases, patterns, and software architectures*. Cambridge University Press, 2011.

[19] D. Greer and Y. Hamon. Agile software development. *Software: Practice and Experience*, 41(9):943–944, 2011.

[20] R. Hebig, T. H. Quang, M. R. Chaudron, G. Robles, and M. A. Fernandez. The quest for open source projects that use UML: mining GitHub. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 173–183, 2016.

[21] A. Ivanchikj, S. Serbout, and C. Pautasso. From text to visual bpmn process models: Design and evaluation. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 229–239, 2020.

[22] D. Karagiannis. Agile modeling method engineering. In *Proceedings of the 19th Panhellenic Conference on Informatics*, pages 5–10, 2015.

[23] O. Kautz, A. Roth, and B. Rumpe. Achievements, failures, and the future of model-based software engineering., 2018.

[24] E. Laurenzi, K. Hinkelmann, and A. Van der Merwe. An agile and ontology-aided modeling environment. In *IFIP Working Conference on The Practice of Enterprise Modeling*, pages 221–237. Springer, 2018.

[25] T. Levendovszky, D. Balasubramanian, A. Narayanan, and G. Karsai. A novel approach to semi-automated evolution of dsml model transformation. In *International Conference on Software Language Engineering*, pages 23–41. Springer, 2009.

[26] D. Marshall and J. Bruno. *Solid code*. Microsoft Press, 2009.

[27] F. Martins and D. Domingos. Modelling iot behaviour within bpmn business processes. *Procedia computer science*, 121:1014–1022, 2017.

[28] J. Metrôlho, R. Araújo, F. Ribeiro, and N. Castela. An approach using a low-code platform for retraining professionals to ict. In *11th annual International Conference on Education and New Learning Technologies*, 2019.

[29] J. Metrôlho, F. R. Ribeiro, and R. Araujo. A strategy for facing new employability trends using a low-code development platform. In *14th International Technology, Education and Development Conference*, pages 8601–8606, 2020.

[30] T. Miranda, M. Challenger, B. T. Tezel, O. F. Alaca, A. Barišić, V. Amaral, M. Goulão, and G. Kardas. Improving the usability of a mas dsml. In *International workshop on engineering multi-agent systems*, pages 55–75. Springer, 2018.

[31] A. Mostashari. *Stakeholder-assisted modeling and policy design for engineering systems*. PhD thesis, Massachusetts Institute of Technology, 2005.

[32] D. Mosteller, L. Cabac, and M. 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.

[33] S. Naujokat, M. Lybecait, D. Kopetzki, and B. 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.

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

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

[36] B. Ramesh, L. Cao, and R. Baskerville. Agile requirements engineering practices and challenges: an empirical study. *Information Systems Journal*, 20(5):449–480, 2010.

[37] S. A. Soares and M. I. Cortés. Elihu: A project to model-driven development with naked objects and domain-driven design. In *ICEIS (2)*, pages 272–279, 2018.

[38] G. C. Sousa, F. M. Costa, P. J. Clarke, and A. A. Allen. Model-driven development of dsml execution engines. In *Proceedings of the 7th Workshop on Models@ run. time*, pages 10–15, 2012.

[39] B. Unhelkar. *Software engineering with UML*. CRC Press, 2017.

[40] D. van der Linden, I. Hadar, and A. Zamansky. On the requirement from practice for meaningful variability in visual notation. In *Enterprise, Business-Process and Information Systems Modeling*, pages 189–203. Springer, 2017.

[41] D. van der Linden, I. Hadar, and A. Zamansky. What practitioners really want: requirements for visual notations in conceptual modeling. *Software & Systems Modeling*, 18(3):1813–1831, 2019.

[42] S. Vaupel, D. Strüber, F. Rieger, and G. Taentzer. Agile bottom-up development of domain-specific ides for model-driven development. In *FlexMDE@ MoDELS*, pages 12–21, 2015.

[43] N. Visic, H.-G. Fill, R. A. Buchmann, and D. 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.

[44] M. Vještica, V. Dimitrieski, M. Pisarić, S. Kordić, S. Ristić, and I. Luković. An application of a dsml in industry 4.0 production processes. In *IFIP International Conference on Advances in Production Management Systems*, pages 441–448. Springer, 2020.

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

A

Anleitung zum wissenschaftlichen Arbeiten

This chapter contains a detailed explanation of the implementation of our tool, a quick start manual, and a tutorial on how to use the tool we developed.

A.1 Components, concepts and implementation

To implement our approach we use the open-source IDE Glamorous Toolkit built and maintained by feenk GmbH.[1] Glamorous Toolkit is built with Pharo, and it enables users to work with a vast amount of different technologies. It is a moldable development environment that contains a source code editor, a software analysis platform, a data visualization engine, and a live notebook. In Glamorous Toolkit, one can create custom visual representations for any kind of data in the development process (see, Figure A.1).

In the subsequent subsections, we take a closer look at the core tools of the Glamorous Toolkit that we used in our approach, *e.g.*, the Playground, the Inspector, and the Coder. Furthermore, we also take advantage of the graphical components available through the Glamorous Toolkit IDE, *i.e.*, Bloc.

At the end of this section, we show how we implemented our tool.

[1] "feenk", accessed 4th December 2021, `https://feenk.com/`

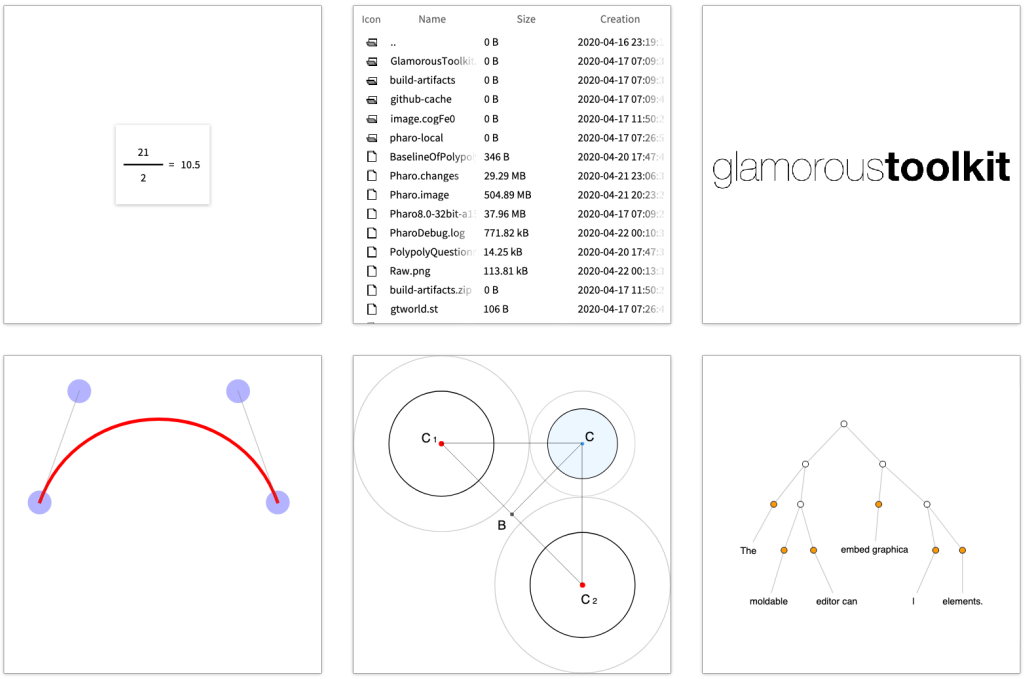


Figure A.1: Different visual representation for different kinds of data (*e.g.*, a fraction, a directory, an image, *etc.*)

A.1.1 Playground

The playground is a core component of Glamorous Toolkit where multiple code snippets can be executed, tested, evaluated, and inspected as seen in Figure A.2. This component can also be embedded in other components. For our tool, this will be the starting point for an empty modeling project or an already existing example domain model. By executing the command `AMTController new` a new canvas appears ready for modeling.

A.1.2 Inspector

The inspector is a tool that contains a multitude of views that opens up on the right when inspecting a code snippet or any other object in the playground. It allows us to define custom views and navigate through every object as seen in Figure A.1. This component will let us inspect all data live of our domain entities along with the modeling and implementing process.

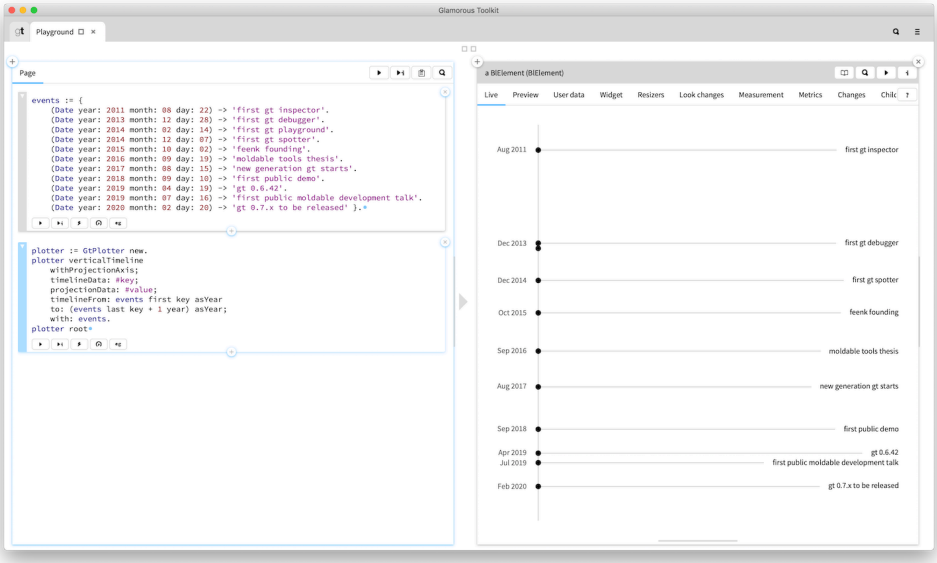


Figure A.2: The playground with two code snippets that can be executed and inspected independently

A.1.3 Bloc graphical stack

The Bloc graphical stack is a vector-based graphical framework on top of which the Glamorous Toolkit is built. This framework enables us to provide any kinds of visual interfaces that we wish to design. The Bloc framework allows us to inspect our visual representations of entities in the inspector as seen in Figure A.1.

A.1.4 Actor model and message propagation

To have a software implementation framework that helps us define behavior and logic within and between domain entities, we chose the Actor model. With the concept of the Actor model, we treat our domain entities as actors. Actors are computational entities. These actors can send, receive and process messages from other known actors and create new actors.

We created each domain entity with the capabilities to send, receive and process messages from known actors. For the message propagation we use a DSML-like code syntax, the implementation of the syntax for sending messages can be seen in Listing 6.

```
1 createAndSendMessageTo: recipient with: data and: type
2     <AMTSend>
3     | message |
```

```
    message := AMTMessageBlueprint new.
    message sender: self.
    message recipient: recipient.
    message data: data.
    message type: type.
    recipient addToMailbox: message
```

Listing 6: Implementation of the DSML-like syntax that can be used by actors to send messages

This syntax can then be later used by domain entities/actors to send messages to any other domain entity (for ATM example code see Listing 7).

```
createAndSend <AMTSend>
    | bank |
    state := state - 200.
    knownActors do: [ :actor | actor class = Bank
        ifTrue: [ bank := actor ] ].
    self createAndSendMessageTo: bank with: '200' and: 'Withdrawal'
```

Listing 7: ATM's use of syntax to create and send a Withdrawal message to a Bank entity

For these messages to have an effect our entities also need methods to receive and process those messages. For example, the Bank entity from our running example needs a definition of behavior. The behavior upon arrival of a withdrawal message from an ATM entity can be seen in Listing 8

```
receiveWithdrawal <AMTReceive>
    |message|
    message := mailbox at: 1.
         self when: (message type = 'Withdrawal' )
              do: [ state := state - message data asNumber].
        self receive
```

Listing 8: Bank's definition of how to behave upon arrival of a withdrawal message

A.1.5 Implementation

For our tool in Smalltalk we used a Model View Controller implementation pattern. So we split our source code package with tags, as is usual in IDEs such as Pharo and Glamorous Toolkit. These tags then contain the needed classes that define our tool.

First of for our `Model` we implmented the classes `AMTModel`, `AMTMessageBlueprint` and `AMTAbstractActor`. The `AMTModel` holds the actors in an `OrderedCollection` and contains further logic to add new actors. The `AMTMessageBlueprint` and `AMTAbstractActor` are classes that we clone to create new actors or create new messages. For the actor blueprint we needed to implement methods for how to initialize our domain entites as actors. This initialization contains logic for a mailbox, for received messages, an address, an `OrderedCollection` for all known actors and a state that can

hold complex information (for exmaple the account balance of the ATM). The message blueprint on the other hand defines data, type, recipient and sender of a message. How the definition of a message is used can be seen in Listing 6.

Our `View` initializes the canvas with the class `AMTView` where domain entities are created, dragged and dropped. Furthermore it also creates the visual elements for the representation of every domain entity. For this we implemented the class `AMTActorElement`. The `AMTActorElement` holds all logic for widgets, object cards and tabs. New widgets, object cards and tabs are defined here.

Lastly the `Controller` only contains one class the `AMTController`. It controls the view and mediates between the model and the view.

For further inspection we suggest the analysis of our source code in our repository.

A.2 Quick start

Take a look at our GitHub repository.[2]

A.2.1 Step 1

Download and open the latest image of the Glamorous toolkit from their official homepage.[3] Open a *Playground* and execute the following script.

```
Metacello new
    baseline: 'AMT';
    repository: 'github://artthik/ActorModelingTool';
    load
```

A.2.2 Step 2

To start with a blank canvas with an initial domain entity execute the following command:

```
AMTController new
```

To start with a simple Hello World example execute following command:

```
AMTHelloWorldExample new
```

To inspect the AMT use case execute following command:

```
AMTExampleATMWithdrawal new completeScenario
```

To inspect the Vostra example execute following command:

```
AMTVostraExample new completeScenario
```

[2] https://github.com/artthik/ActorModelingTool
[3] https://gtoolkit.com/

A.3 Simplified tutorial to model the ATM domain in our tool

A.3.1 Create domain objects visually

Our example with the ATM needs multiple domain objects, which need to be created first. The *ATM* object needs to communicate with a *bank* object to retrieve the *account* information. Account information in our case would be the balance and the user credentials which will be validated by the ATM. All of these domain objects (*i.e.*, ATM, bank, and account) can be created visually by any stakeholder. These empty domain objects can then be positioned in the tool as needed.

A.3.2 Implement the object behavior in views

Once the domain objects are created, the technical stakeholders can use the default views (*i.e.*, Main and All methods), change or create new views needed for all the stakeholders. For the developers' needs, they can use the *All methods* default view. In this view, software developers can implement domain object behavior and logic as needed. The implementation needs to be abstracted to fit the actor model with message propagation. By sending, receiving, and processing messages all behavior can be depicted. In this step, the software developer can also implement unit tests and use case *example methods* for further prototype execution and testing.

A.3.3 Execution and testing the prototype

With objects created and behavior implemented, simple use cases can now be tested and visualized in the tool. Any stakeholder can step through each process step of each domain object and audit the created domain model. Further changes can then be iteratively and incrementally applied by all stakeholders.