FAKULTÄT FÜR !NFORMATIK

UML Diagram and Element Generation Exemplary Study on UMLet

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Johannes Pölz
Matrikelnummer 0225722

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Mag.rer.soc.oec Stefan Biffl

Wien, February 3, 2009 (Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Johannes Pölz, Rauscherstr. 7/10, 1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit—einschließlich Tabellen, Karten und Abbildungen—, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, February 3, 2009

Zusammenfassung

Flexibilität, Effizienz und Usability sind die großen Ziele der Software Entwicklung. In dieser Diplomarbeit präsentieren wir mehrere Verbebesserungen des UML Zeichentools UMLet. Solche tools werden in den frühen Stadien des Softwareentwicklungsprozesses verwendet—dort wo ein schnelles erstes Design benötigt wird. Der größte Konkurrent ist noch immer Bleistift & Papier, weil—obwohl es einige Nachteile hat—es noch immer die größte Flexibilität und hohe Effizienz bietet.

Um auf dem UML Tool Markt konkurrenzfähig zu bleiben muss UMLet vom Benutzer erweiterbar sein. Deswegen stellen wir einen Erweiterungsmechanismus vor, der dem User gestattet eigene UML Element per *End User Development* zu erstellen. Das erhöht die Flexibilität, weil der User nicht auf ein vordefiniertes Set von UML Elementen eingeschränkt wird.

Zusätzlich stellen wir einen neuen Ansatz ganze UML Diagramme zu generieren vor. Dieser Ansatz generiert das Diagram aus einem Text, den der User eingibt. Das heißt, dass der Nutzer in der Lage ist mit der Tastatur ganze UML Diagramme zu erstellen—was die Effizienz stark erhöht.

Am Ende der Arbeit werden noch einige Refactorings, Verbesserungen der Usability und neue Features der UMLet Architektur vorgestellt.

Abstract

Flexibility, efficiency and usability are major design goals in software development. In this thesis we present several improvements of the UML drawing tool UMLet. Such tools are usually used in early stages of the software development process—where a first quick design is needed. Their greatest competitor is still pen & paper, because—although having several disadvantages—it provides the greatest flexibility and is very efficient.

To compete on the UML tool market, UMLet has to provide a possibility to be extended by the user. Therefore we provide an extension mechanism, that enables the user to create customized UML elements by *End User Development*. This increases flexibility because the user is not limited to choose from a predefined set of UML elements.

Additionally a new approach to generate whole UML diagrams is introduced. This approach generates the diagram out of user entered text. That means that the end user is able to generate the diagram by using just the keyboard, which increases efficiency by far.

Finally several necessary refactorings, improvements of usability and new features of the UMLet architecture are presented.

Acknowledgements

First of all, I want to thank Emma and Leopold Pölz, my parents, for sponsoring my education as well as giving support all my life.

In addition, I owe special thanks to Martin Auer and Stefan Biffl for guidance and supervision during the creation process of this thesis. Without their feedback and contribution, this thesis would not have been possible.

Furthermore, I want to thank Kathrin Apfelthaler for giving moral support and providing new inspiration whenever needed.

Finally, I want to thank Josef Pfleger for extensive proofreading.

Contents

List of Figures

List of Tables

Listings

1. Introduction

1.1. Motivation

Drawing UML diagrams is a common task during software engineering studies. Most of the time modeling in UML has been time-consuming—compared to drawing the diagrams with pen and paper. The user interfaces of many UML tools are not intuitive. A large amount of the working time on UML modeling is used to find workarounds for UML tool peculiarities. This is especially demotivating as the real solution for the problem usually could be found in a fraction of the time needed to implement the solution in the specific tool.

Therefore searching for an unrestrictive and simple UML tool is unavoidable, when confronted with UML design. Fortunately the freeware application UMLet—although still not perfect—is not as restrictive as other UML drawing applications. The ease of use, flexibility, and efficiency of this freeware application already is quite above-average. Therefore the main goal of this thesis is to enhance the flexibility and efficiency of UMLet by improving the user interface transparency, allowing the end users to create new UML elements on-the-fly, and better integrating the tool in standard development environments.

1.2. UML

The *Unified Modeling Language* (UML) [6] is widely used for modeling object-oriented software systems. It aims to cover most parts of the software development process and has been applied in various environments (e.g., real-time systems [13], or decentralized production control systems [21]). It was developed between 1994 and 1997 by the "3 Amigos": Jim Rumbaugh, Ivar Jacobson, and Grady Booch. Version 1.1 was standardized in November 1997 by the *Object Management Group* (OMG) [34]. The OMG is a non-profit industry group now responsible for defining and maintaining the UML specification. The next major revision—UML 2.0—became the standard in October 2004 and since then evolved to the current version 2.1.2 of the UML superstructure and infrastructure specifications.

Diagram Type	Description
Activity Diagram	Models high-level processes, including data flow of complex processes within a system
Class Diagram	Describes classes, their variables, functions, and relationships
Communication Diagram	Visualizes the communication/message flow between classes
Component Diagram	Describes the components of a system as well as their interactions and interfaces
Composite Structure Diagram	This diagram is used to model the cooperation between class instances over communication links
Deployment Diagram	This diagram is used to model the hardware used to deploy different software components as well as the associations between them
Interaction Overview Diagram	A variant of the Activity Diagram which contains other Interaction Diagrams instead of only activities
Object Diagram	Describes objects at a specific point of time. This behavior is needed to describe special cases of class or communication diagrams
Package Diagram	Shows the packaging of different model elements as well as the dependencies of the different packages
Sequence Diagram	Models the sequential behavior of a part of the system
State Machine Diagram	Describes the different states of an object as well as the possible transitions between states
Timing Diagram	Describes the states of an object over time
Use Case Diagram	Describes the different use cases and possible actors of software systems or parts of it

Table 1.1.: UML diagram types

UML has unified and standardized the concepts of previous graphical notations like Booch or OMT. It currently features six types of structure diagrams, as well as seven types of behavior diagrams. All in all, UML defines more than 130 elements [35]; new ones are being added in each new version. A short description of the different diagram types is shown in table 1.1.

1.3. UML Tool Requirements

Due to the widespread use of UML, a large number of tools is available. According to Smith [30], these can be categorized into:

- UML drawing tools;
- code-centric tools; and
- framework tools.

UML drawing tools focus on fast diagram sketching and offer great flexibility. Code-centric tools restrict the UML specification to fit specific programming languages, but provide additional features like code generation and reverse engineering. Framework tools go one step further by supplying the user with extended code generation mechanisms as well as providing automated test case generation.

One of the most important requirements for all of these tools is the conformity to standards. These standards include the most common application behaviors like keyboard shortcuts, toolbars, and copy / paste functionality. Besides these usability features every application should provide several export possibilities, like PDF exporting or printing. Although all these features seem rather unimportant they result in a significant benefit for the end user.

Another benefit of standard compliance is that most users are already familiar with the graphical user interface and do not need special training to use all of the interfaces features. This saves time for both developers and users. Developers need not to describe the basic tool behavior in a manual (ideally there is no need for a manual) and users do not get frustrated by reading through instructions for hours.

Furthermore UML tools should:

- provide great flexibility to the end user—even allowing non standard notations
- be efficient to use
- be easy to learn

Although these requirements exist for all UML tool categories they are especially important for UML drawing tools, because those usually do not provide source code generation and thus no additional benefits—besides documentation—are gained for the software development project. Because of this fact the tool has to be as efficient as possible, has to provide a maximum of flexibility because no strict UML standard compliance is needed, and should be very easy to learn to reduce the overhead costs. In an ideal case the work with the UML sketching tool is as flexible, as easy and even more efficient than drawing the diagram with pencil and paper.

1.3.1. Flexibility

Flexibility is one of the most important features of UML drawing tools. Although the UML standard must be supported, UML drawing tools are able to additionally support non UML compliant elements to satisfy special design needs of users. This is possible because no further transformations are required to the diagram. In an ideal case all elements the user needs are included, or the user is enabled to create his own elements. Enabling the user to paint almost all elements he wants is an important feature, because the tool has to replace diagram drawing with pencil and paper.

Providing every user with all diagram elements she needs has three disadvantages:

- Increased development costs for the tool supplier
- Developers might not know all elements required by a specific user
- Users are confronted with a too large pool of elements

These problems can be avoided, if the user is enabled to design customized elements. New users are then confronted with a reasonable amount of elements and advanced users are able to extend the tool—satisfying their special needs.

1.3.2. Efficiency

UML drawing tools are usually used for design purposes in the early stages of the software development process. Efficiency becomes even more important than it usually is, because the tool has to compete with the pencil and paper approach.

One problem with increasing efficiency is, that optimizations for specific diagram drawing processes may limit the user's flexibility (e.g. connections that snap in when near an element make connecting lines to elements easier, but remove the option to place a line near an element without connecting them). Because of this interdependency, every application has to be well balanced between flexibility and efficiency, to satisfy all users needs.

1.3.3. Easy to Learn

Getting started with a tool without reading manuals has become a standard. Most users are already accustomed to tools that do not need other learning effort than just using the tool. Therefore this feature is most important to acquire new users.

Although the goal of easy learning sounds easy, finding a solution is one of the most challenging tasks in software development.

1.4. UMLet

UMLet is a UML diagram sketching tool that focuses on keeping diagram sketching simple and efficient. The UMLet project has been initialized by Martin Auer[1] and has been further developed by several students of the *Vienna University of Technology*.

UMLet focuses on enabling the user to draw diagrams efficiently instead of supporting the user with a vast amount of features like most UML framework tools (e.g. IBM Rational Rose[2]). Those framework tools try to support the user during the whole development process—from code generation to back-transformation of code to UML. One drawback of such tools is that they need to restrict the user's freedom of diagram drawing to ensure that all UML standard properties are met. The standard has to be followed to allow the tools transforming the diagrams into other forms (e.g. source code).

UMLet goes a different path and tries to enable the user to draw diagrams as unrestrictive as possible. It therefore does not support additional modeling features like code generation. Its keys—which are consistent with the UML Tool requirements of chapter 1.3— to success are:

- Flexibility
- Efficiency
- Ease of use

To achieve flexibility UMLet does not restrict the user to the UML standard but allows the user to combine elements in whatever manner she likes. Therefore the user is even enabled to draw non UML diagrams as long as the corresponding elements exist. The absence of restrictions does not only increase the flexibility of the application. It also increases its usability, because the user does not have to be familiar with UML standards to be able to get started with the application. Therefore UMLet is especially suitable for UML beginners.

Efficiency is achieved by UMLet's simple design. As shown in figure 1.4 an element's properties are edited within a single text field instead of a complex arrangement of element properties.

[1]`http://www.auer.net`
[2]`http://www-306.ibm.com/software/awdtools/developer/rose`

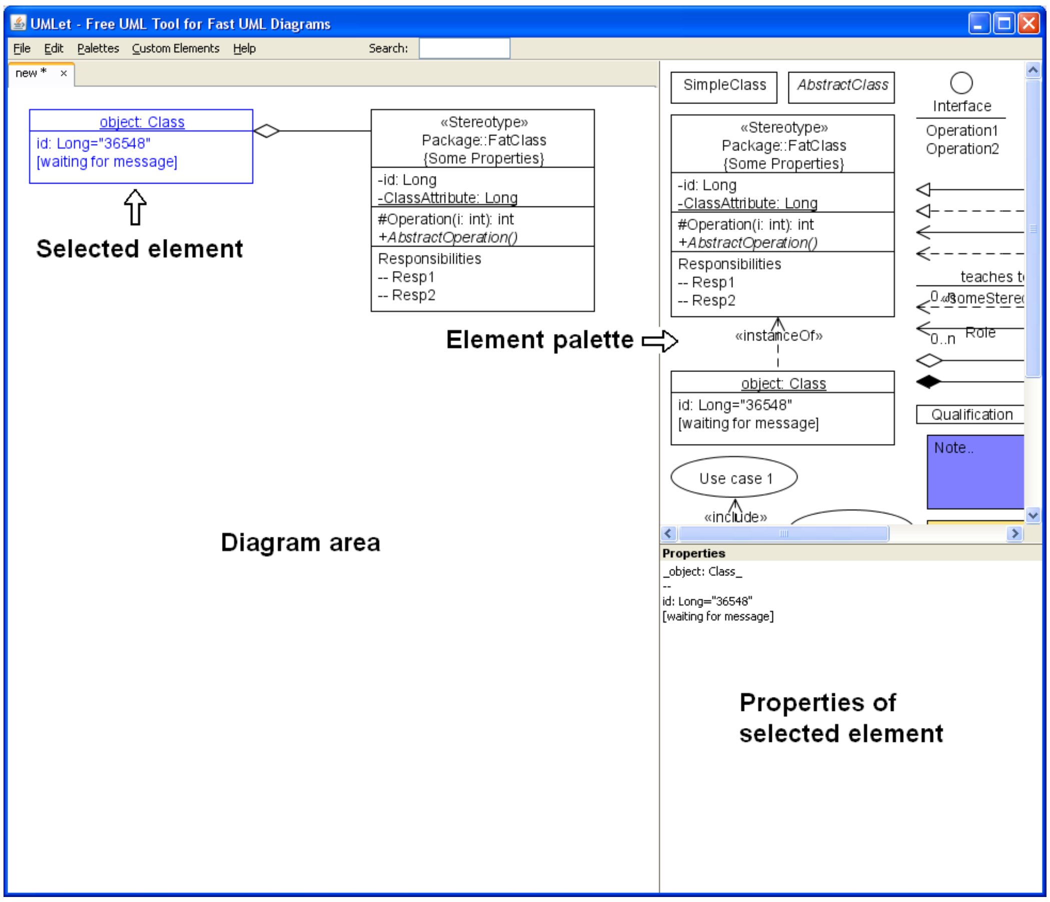


Figure 1.1.: UMLet screenshot

Therefore each property can be accessed without requiring navigation to a sub-menu or anything similar. Additionally UMLet provides several other features that increase efficiency, like easy element duplication (by double clicking an element) or enabling the user to generate whole diagrams by text.

Ease of use is—like efficiency—achieved by simplicity. The design does not contain any *hidden* features. Every feature is presented right in the front. Additionally UMLet concentrates on the core features of diagram editing—e.g. adding, moving, editing, and resizing elements—to avoid unnecessary complexity. Another startup help is that the core features are presented in the property panel on application startup and whenever a new diagram is created. The user does not need to search through a manual to get the information.

Although UMLet already is a great tool, there is still much room for improvements. During this thesis some major and many minor improvements will be presented. Within the thesis UMLet 9.x will be referred to as UMLet, because this is the currently newest version of the program. To get a quick overview about the features of UMLet you may download the executable program and the source code at `http://www.umlet.com`.

1.5. Structure

The work is divided into four main chapters. Each of the chapters defines its own goals of the research on UMLet.

To increase flexibility, chapter 2 introduces an approach to create custom elements in a simple and flexible way. Several possibilities how this could be achieved are discussed and finally detailed information of the implemented solution is provided.

To increase efficiency, chapter 3 proposes an approach to generate whole UML diagrams from the text entered into a single text field. Several advantages, disadvantages, and variations of this approach will be discussed and finally the approach's performance will be compared to the usual drag & drop diagram editing approach.

Chapter 4 combines the approaches of chapter 2 and chapter 3 by adding the possibility to create custom elements for textual diagrams. This enables the user to increase the efficiency of using created custom elements.

To increase the ease of use, several usability improvements and refactorings to UMLet's software architecture are presented in chapter 5. The description of UMLet's architecture gives a general implementation background of UMLet and should therefore especially support further development.

2. Flexible Software Customization via User-Defined Source Code Fragments

"Using a computer" evolved considerably over the last decades from an end-user point of view. In the early days, end users had to actually program the computer, which provided only a basic set of functionality. Soon, of course, some of those early programs evolved into ever more sophisticated special-purpose applications to be distributed to an ever-growing set of potential end users. These applications provided a specific, constrained set of high-level functionality (e.g., the Unix set of command line tools), while relieving end users from dealing with low-level programming. This process towards specialized, stand-alone applications was consolidated with the advent of graphical user interfaces (GUIs). These apps provide higher-level functionality (e.g., a word processor), and they are arguably easier to use for a wider audience. However, the flexibility and intra-tool communication is further restricted: the end user can no longer tweak the tool thoroughly.

This outline—from basic, general-purpose programming systems to less flexible, high-level custom applications—greatly simplifies the manifold currents in the evolution of software, its application and user base, and its flexibility. Several counter-trends can be readily identified:

1. Many modern applications (Eclipse, Excel, Firefox, to name but a few) can be enhanced and modified using plugins. The underlying techniques differ; the approaches are similar: in an external tool some partial functionality is defined and packaged, and then attached to the main application.

2. A variety of tools allows to add user-defined functions or statements. OLAP applications, for example, let users enter SQL statements if the usual drag-and-drop GUI can't easily assemble particularly complex queries. Financial applications like cash-flow engines and pricing tools let users enter additional business logic and payoff functions—basically additional mathematical formulas and functions. Usually, the end user is allowed to access some basic set of mathematical functions, to which some restricted data structures are

exposed. The modifications can usually be done within the tool, at run time.

3. On a larger scale, applications or operating systems can provide a whole macro language or framework (VBA for MS Office, AppleScript..). In this discussion, we'll focus on VBA-like macro languages due to their wide-spread use. This approach exposes more of the main application's functions and data structures to the end user. The macro language's scope, however, is often still restricted to a subset of the internal APIs, in part because the language is usually simpler that the application's core language (e.g., Basic vs. C++). Often, inconsistencies arise: see, for example, how Excel's native cell formulas like *MMult* are accessed within VBA. Advanced operations are possible, but often require embedding externally created library functions (in VBA, with the *Declare Function* command).

4. Finally, open source software discloses an application's entire source code. End users can inspect and modify all aspects of the software. Users, particularly large institutions, regularly adapt open source software to their needs, e.g., database software like PostgreSQL is modified to handle new data types.

Approach (1) is appealing—our main critique is the media break: the plugin is developed externally and must be included explicitly in the main application. In many instances, it is preferable to allow run-time modifications seamlessly, within the application, like in approach (2).

Approach (4) is geared towards power users; other end users might be overwhelmed by the setup and sheer complexity of the code. Too much is exposed—many parts of the code are unlikely to be modified. The highest degree of flexibility is achieved, but at the expense of severely reducing the target audience.

This chapter describes an approach of providing seamless flexibility for many end users, by letting users alter parts of the core source code at run time. Conceptually, it lies between approach (2) and approach (3):

- Users can access the full core API and data structures of the tool, unlike in (2). Not only mathematical functions, but the graphical behavior and the GUI's reaction to user input can be modified.

- Users can use the tool's core language (Java in our case), not a stripped-down or simplified macro dialect. API calls and data access are thus consistent in the user-defined code parts and the surrounding original code, unlike in (3). This avoids redundancies in the interfaces, data structures, and their documentation; it is also preferable, for example, if the code should later be included in the official distribution branch of the application.

- There is no media break, i.e., no need for external tools or explicit imports like in (1), or for external DLLs like in (3). Such media breaks are tedious, time-consuming and a prime source for errors.
- While the end user has access to the full range of the internal API and data structures, he can modify only those parts of the code that are expected to require this additional level of flexibility. The proposed approach thus resembles "sand box", as opposed to the "sand bucket" in (2) or the "sand mine" in (3).

This chapter addresses two target audiences: end users and developers of flexible software applications. To end users we present where and how UMLet provides flexibility. To developers we outline the implementation setting and some technical details. We argue that users should demand this increased level of flexibility, not only in a setting where they are allowed to define mathematical expressions, but in ones with more complex tool behavior as well. Developers, in turn, should embrace this way of deferring design decisions to later stages, up to the end user. Often, they themselves will find that developing new tool features is best done at run time, within the tool.

Section 2.1 describes this chapter's goal in greater detail. Section 2.2 gives on overview on related work. Section 2.3 outlines implementation issues. Section 2.4 presents several examples of UMLet's end-user development process. Section 7 concludes and points out future research directions.

2.1. Goals and Approach

The main goals of this chapter are:

- (G1) Enable users to generate customized elements.
- (G2) Avoid a media break when creating custom elements within UMLet–the custom element interface should be provided as part of the main application.

- (G3) Provide full access to modify the elements' behavior, instead of a stripped-down macro language.

UML defines a wide variety of graphical elements; each new UML version modifies them, or adds new ones—so there is a constant need for UML tool updates. End users can thus either wait for a new version of a given UML tool, or design their own UML elements. This chapter proposes a way how end users can develop such new graphical elements within the UML tool UMLet [2] (G1).

Following Ruyter's suggestions [28], the goal is to keep the approach simple and to provide immediate feedback to the user—see section 2.1.1. The way users create new elements is that they can specify—via Java code—how new elements are painted. This definition takes place within the tool UMLet, at run time (G2). To reduce the complexity of this user generated code we use a code frame or template to avoid exposing unnecessarily complex objects to the user—see section 2.1.3.

We use Java as end-user programming language. The main reason for choosing Java is the greater flexibility compared to a custom domain specific language (DSL). This allows the user to access both some methods pre-defined for user convenience, as well as Java's complete set of graphics and core libraries (G3). Furthermore, Java's reflection features make the implementation of this additional degree of flexibility straight-forward. Another reason for using a popular object-oriented language is that many UML tool users are already familiar with it.

2.1.1. UMLet's User Interface

UMLet's user interface (figure 2.3) basically provides three panels - the diagram panel, the palette panel, and the property panel. The diagram panel, of course, displays the diagram and all the UML elements it contains, and lets the user modify their location—just like other graphical tools. The palette panel exposes all available elements and lets users add instances of those elements to the diagram panel (it does not represent elements by icons; instead, it represents them like they would look on the diagram). Finally, the properties panel is a text panel that lets users view and modify the properties of a UML element.

In most UML tools, element properties are modified via pop-up windows, where users can change element attributes. The property panel in UMLet provides a shortcut to this workflow: it displays—as a single string—all relevant properties; users can fast and easily modify the element's properties by changing this string. For example, to add new methods to a UML class element, one can simply enter their names as several new lines to the string given in the property panel.

For each UML element, UMLet's display or drawing logic is simple: the string given in the property panel is interpreted, and the graphical representation of that UML element is modified according to this interpretation. More specifically, all UML elements on a diagram correspond to a Java object that "draws itself", i.e., interprets its own property panel string and modifies its own graphical representation.

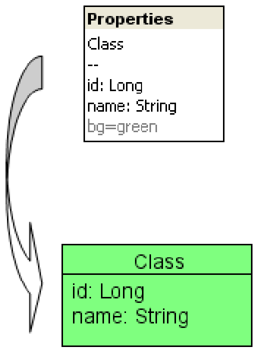
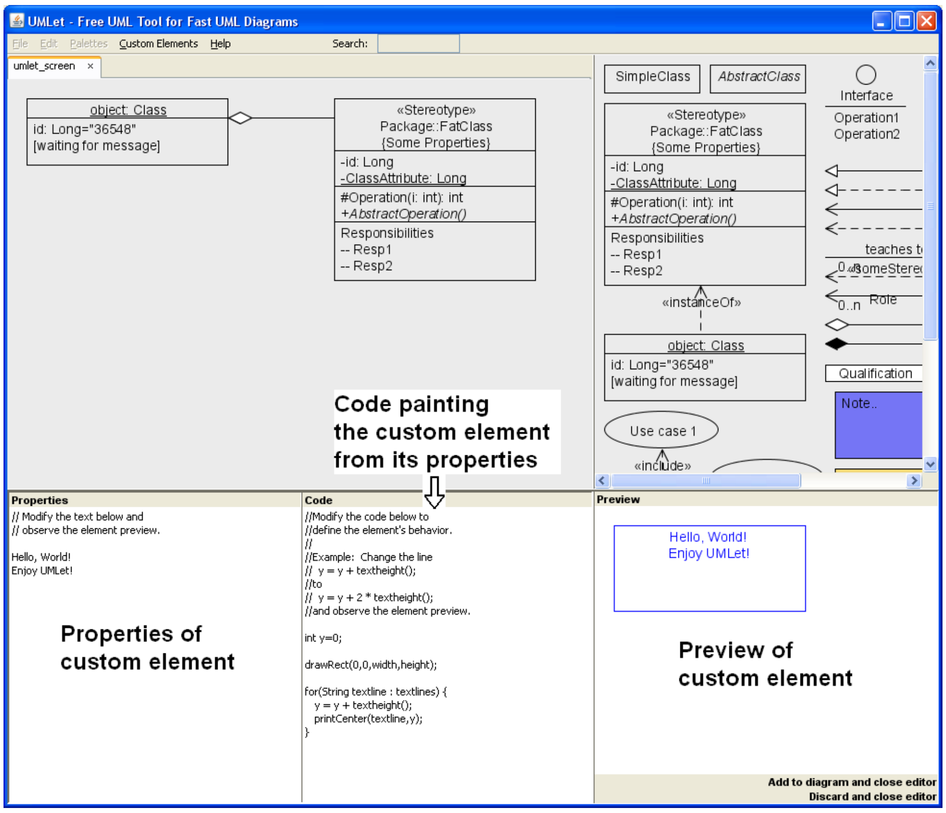


Figure 2.1.: Property text parsing example

For example, in figure 2.1 the property panel's text of a UML class element is interpreted by that class element to draw itself. The first line is interpreted as the class name and printed on top of the class, aligned centrally. Subsequent lines are treated as method or attribute names, which are printed left-aligned. The string "– –" (a double dash) is interpreted as a horizontal line separating class name, attribute names, and method names; the string "bg=green" sets the element's background color to green.

2.1.2. Custom Elements

To keep the development of new custom elements simple, the development environment has to be integrated into the application itself. Integration via a separate window or by providing a plugin interface would be possible, but it would represent a media break and an additional barrier for end users. A more direct way is provided: as soon as a new custom element is created, the property panel expands into three panels. The panels shown in figure 2.2 are (from left to right) the property panel, a source code panel, and a preview panel. The user can now modify the custom elements on two different levels. First, as before, he can change the text in the property panel and see the consequences in the graphical representation in the preview panel. But in addition, in the source code panel he can change the source code that is responsible


UMLet - Free UML Tool for Fast UML Diagrams
File Edit Palettes Custom Elements Help
Search:
umlet_screen
Code painting the custom element from its properties
Properties of custom element
Preview of custom element
Properties
Code
Preview
Hello, World!
Enjoy UMLet!
Add to diagram and close editor
Discard and close editor

Properties:

```
// Modify the text below and
// observe the element preview.

Hello, World!
Enjoy UMLet!
```

Code:

```
//Modify the code below to
//define the element's behavior.
//
//Example:  Change the line
//  y = y + textheight();
//to
//  y = y + 2 * textheight();
//and observe the element preview.

int y=0;

drawRect(0,0,width,height);

for(String textline : textlines) {
   y = y + textheight();
   printCenter(textline,y);
}
```

Figure 2.2.: UMLet screenshot - custom element UI

for interpreting the property panel text and for drawing the element in the preview panel. At the same time, thus, a user can change an element's properties, and how those properties are transformed into a graphical representation.

The code is compiled continuously in the background, and immediate feedback is provided to the end user by re-drawing the element in the preview panel. If the code contains errors, the affected source code lines are highlighted. Several examples of simple custom elements are shown in section 2.4.

2.1.3. Code Template

End users can thus modify the Java code that determines how a UML element properties are converted to that UML element's graphical representation. To avoid exposing to much application logic to the end user, UMLet provides a code template that hides most of the the UML element's internal structure (i.e., its Java class definition) from the user. Now, the user only has to modify the source code for a single method—the drawing method of the graphical component. Implementing a single method rather than a full class is easier because it does not require background knowledge of the object-oriented architecture; in addition, most surrounding or support functionality—like file dialogs, context menus, file exports—are unlikely to need change, and would only be a distraction.

Another simplification is that the user does not need to worry about color settings and command parsing. In general, this is done by a global parsing method. The template takes care of applying the global parsing mechanism like the color settings, or the default behavior of element resizing. This way, some settings are consistent for all, even newly created, elements. Only properties specific to a new element have to be processed by the user's code.

Whenever the user's code needs to be compiled, it is inserted at the appropriate position in the template class, and the entire class is compiled. If successful, the element is generated (i.e., an object of that class is instantiated), and put on the preview panel, where it draws itself using the user-provided drawing method.

2.2. Related Work

2.2.1. End-User Development

The concept of *end-user development* (EUD)—defined by Beringer as a "design problem to enable subject matter experts to create or modify executable software components" [3]—has been investigated for several decades [24]. However, as several authors notice, the advances are behind expectations [32, 14]. Beringer, for example, notes that EUD applications "only realize a fraction of EUD's potential and still suffer from several flaws" [4]. He lists only a handful of partially successful applications, like macros, spreadsheets, and email filters.

The need for EUD applications, however, becomes more evident as increasing numbers of domain experts need to customize their software tools to ever higher degrees, preferably at "use time" [20]: they become "unwitting programmers" [10] as they evolve from customizing software to creating new software functionalities. Fischer identifies an even bigger group of end users, the "much larger population of intellectually disenfranchised knowledge workers who are forced into a consumer role" as opposed to the traditional "population of elite scribes who can act as designers" [15].

Several papers describe the requirements for successful EUD applications: Ruyter, for example, stresses that such applications should be simple, give proper user feedback, and motivate the end user to play with the system [28]. Segal, on the other hand, focuses on the problems such approaches need to deal with and overcome, especially the problems of sharing knowledge and of software reuse in a EUD context [29]. Also, common important software engineering activities (like documentation) are unlikely to be performed by the task-focused end users [29]. The "conflict between complexity and power" [14] leads Sutcliffe to a cost-benefit analysis of EUD [32]. Others, like Heng [17], emphazize the need for more communication between system developers and end users to achieve flexible and maintainable systems.

Another issue—the possible technical frameworks underlying EUD applications—is discussed, for example, by Paterno, who states the need for a transformation between intuitive representations of functionality to more precise, but more difficult to develop, ones [5]. Similarly, Fischer describes some frameworks (like the high-level language Java) as having a high cost of learning, and thus not being an ideal EUD environment [14].

Several organizations have recently focused on this topic, for example, EUD-Net[1], EUDISMES[2], EUSES[3].

2.2.2. UML

The *Unified Modeling Language* (UML) [6] is widely used for modeling object-oriented software systems. It aims to cover most parts of the software development process and has been applied in various environments (e.g., real-time systems [13], or decentralized production control systems [21]). It was developed between 1994 and 1997 by the "3 Amigos": Jim Rumbaugh, Ivar Jacobson, and Grady Booch. Version 1.1 was standardized in November 1997 by the *Object Management Group* (OMG) [34]. The OMG is a non-profit industry group now responsible for defining and maintaining the UML specification. The next major revision—UML 2.0—became the standard in October 2004 and since then evolved to the current version 2.1.2 of the UML superstructure and infrastructure specifications.

UML has unified and standardized the concepts of previous graphical notations like Booch or OMT. It currently features six types of structure diagrams (e.g., class diagram, deployment diagram..), as well as nine types of behavior diagrams (e.g., activity diagram, use case diagram..). All in all, UML defines more than 130 elements [35]; new ones are being added in each new version. This puts some strain on UML tools—they often do not cover the whole range of the graphical notation, and need to be continuously updated to reflect new UML versions.

[1] `http://giove.isti.cnr.it/eud-net.htm`
[2] `http://www.eudismes.de`
[3] `http://eusesconsortium.org`

Due to the widespread use of UML, a large number of tools is available. According to Smith [30], these can be categorized into:

- UML drawing tools;
- code-centric tools; and
- framework tools.

UML drawing tools focus on fast diagram sketching and offer great flexibility. Code-centric tools restrict the UML specification to fit specific programming languages, but provide additional features like code generation and reverse engineering. Framework tools go one step further by supplying the user with extended code generation mechanisms as well as providing automated test case generation.

An overview of many UML tools is available on Jeckle's Web site[4]. IBM Rational Rose[5] and Visual Paradigm[6] are two commonly used commercial framework tools that attempt to support a host of development procedures, as well as code generation and reverse engineering. In contrast, the open source tool UMLet[7] [2] concentrates on fast diagram sketching [1] and flexibility. Another aspect of UMLet—presented in this chapter—is the adaptability to new element types via *custom elements*, where users can define new UML element types at run time.

2.2.3. Reflection

Reflection is the ability of a programming language to query and operate with language meta-information, e.g., to get a list of all available members of a class in an object-oriented framework. Cazzola [8] describes its properties of "transparency, separation of concerns, and extensibility" as "accepted as useful for software development and design" and as "likely to be of increasing relevance in the modern software engineering scenario".

Reflection is especially useful to delay parts of a software design from compile time to run time. This property is valueable for addressing flexibility issues in modern software development—it is applied in COTS components (e.g., Hibernate[8]), in plug-and-play supporting software (e.g., Firefox[9]), and in highly customizable software (e.g., UMLet).

[4] `http://www.jeckle.de`
[5] `http://www-306.ibm.com/software/awdtools/developer/rose`
[6] `http://www.visual-paradigm.com`
[7] `http://www.umlet.com`
[8] `http://www.hibernate.org`
[9] `http://www.mozilla-europe.org/en/firefox`

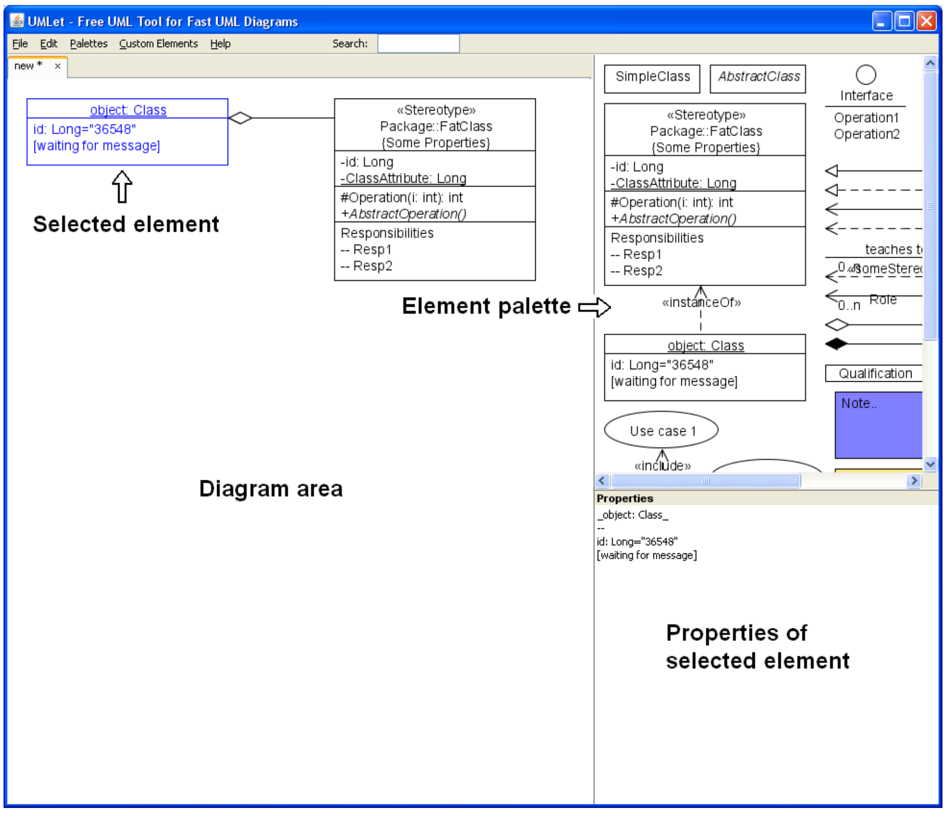


Figure 2.3.: UMLet screenshot

A prominent example of a language providing reflection capabilities is Java. An established application of reflection concepts in the Java environment is, e.g., the Enterprise JavaBeans component technology[10].

2.3. Implementation

The following sections gives a brief implementation overview of the approach presented in section 2.1: it describes the workflow to create a new custom element, the available predefined methods, the saving of custom elements, error handling, and code completion. Finally, some security issues are discussed that arise when custom elements are exchanged between users. Although UMLet is implemented in Java, the implementation details are presented—when possible—in a language-independent way. The approach should be readily applicable in

[10]`http://java.sun.com/products/ejb`

other programming languages that support reflection, provide a compilation API, and some sort of security manager.

2.3.1. Element Generation

The general workflow of the element generation is shown in figures 2.4 and 2.5.

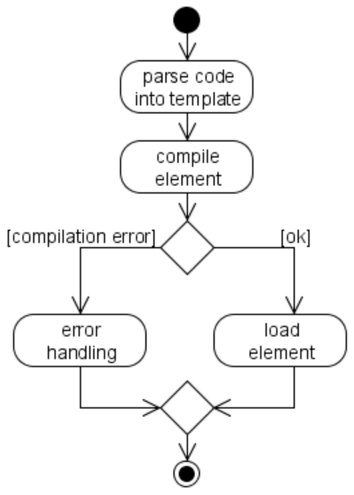


Figure 2.4.: Compilation process

First, the class *ConcreteCustomElement* is generated by inserting the user code into the *paint* method of a template class that extends the class *CustomElement*. Then the new element class is compiled, loaded, and instantiated via the Java class loading mechanism. If a compilation error occurs, an error element is returned instead. After a successful compilation the generated element may be used just like predefined elements.

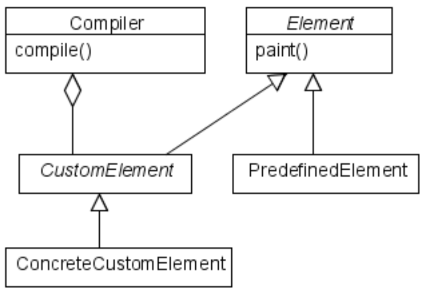


Figure 2.5.: Compilation class diagram

2.3.2. Class Template API

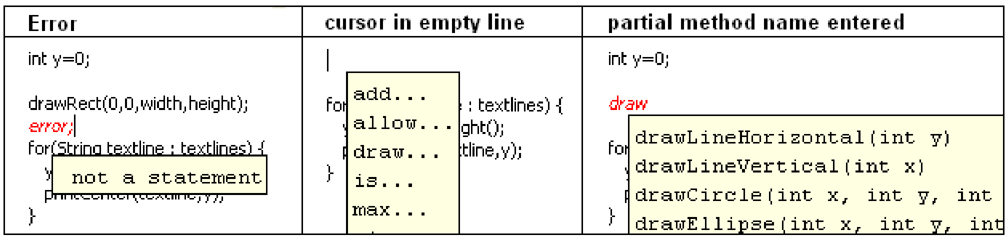
The template class provides several predefined methods. Since the entire Java API is still available, these methods are not a restriction for the end user but rather a help to get started.

Text Drawing Methods

The text drawing methods provide the user with drawing functionality such as text printing and boundary calculation. Text can be printed at either a fixed position or at a fixed vertical position with an alignment relative to the element's horizontal bounds.

The *textheight* and *textwidth* methods return a text line's height and width in pixels.

```
void print(String text, int x, int y)
void printLeft(String text, int y)
void printRight(String text, int y)
void printCenter(String text, int y)
int textheight()
int textwidth(String text)
```



```
int y=0;

drawRect(0,0,width,height);
error;
for(String textline : textlines) {
```

```
not a statement
```

```
|

for
    : textlines) {
    ght();
    tline,y);
}
```

```
add...
allow...
draw...
is...
max...
```

```
int y=0;

draw

for
}
```

```
drawLineHorizontal(int y)
drawLineVertical(int x)
drawCircle(int x, int y, int
drawEllipse(int x, int y, int
```

Figure 2.6.: Compilation panel

Drawing Methods

The predefined drawing methods just call some of the Java AWT package's drawing methods. The advantage when using these methods is that they automatically use a consistent back- and foreground color handling.

```
void drawRect(int x, int y, int width, int height)
void drawLine(int x1, int y1, int x2, int y2)
void drawLineHorizontal(int y)
void drawLineVertical(int x)
void drawCircle(int x, int y, int radius)
void drawEllipse(int x, int y, int radiusX, int radiusY)
void drawPolygon(Polygon polygon)
```

Resizing Methods

Resizing methods help the user to handle elements that dynamically resize themselves. If the *allowResize* method is invoked with the value *false*, the user will be unable to resize the element manually. In this case the element's dimensions have to be set in the—user-provided—*paint* method.

The *setElementCentered* method moves the element's anchor point to its horizontal center. Any update to the element's width is then applied in equal measure on both sides. If the element resizable—i.e., the *allowResize(false)* method has not been called—the *setElementCentered* method has no effect.

The *isManualResized()* method enables the handling both manual and automatic resizing. The element's dimensions still have to be computed in its paint method but only if *isManualResized()* returns *false*. It returns *true* if the user has already manually resized the element—in this case, automatic resizing is suppressed. To switch back to automatic resizing, the user has to remove the property *autoresize=false* that is automatically added to the element's properties as soon as the user manually resizes the element.

```
void allowResize(boolean allow)
void setElementCentered()
boolean isManualResized()
```

Other Methods

The *addStickingPoint* method alters the element's *sticking polygon*. The sticking polygon specifies where exactly UML relations (basically, the arrows in a diagram) stick to the element and if they should be repositioned when a UML element they stick to is moved by the user. The sticking polygon defaults to a rectangle enclosing the entire element.

The *min* and *max* methods return the minimum or maximum of its parameters. This is a shortcut to Java's *Math* package to provide faster access to these frequently used methods.

```
int min(int value1, int value2)
int max(int value1, int value2)
void addStickingPoint(int x, int y)
```

2.3.3. Error Handling

The error handling mechanism (figure 2.6) for showing compilation errors in the code follows standard IDE conventions: it takes the generated compiler error messages and marks the erroneous lines. If the user moves the mouse over those lines, the error message is displayed.

2.3.4. Code Completion

Many popular development tools (e.g., Eclipse) implement a code completion feature, where a list of all available methods and variables is displayed as the user types. In order to provide the user with a quick overview of the available predefined methods (see section 2.3.2), the method prefixes are displayed when the user moves to an empty line (figure 2.6); as soon as the user starts typing, a detailed list of methods and variables matching the user input is displayed.

2.3.5. Saving and Sharing an Element

There are several ways to save and distribute a custom element; they have different advantages and disadvantages:

1. The custom elements can be stored in separate files, or within the UML diagram file they are part of.
2. The custom elements can be stored as Java code, or as compiled class file.
3. If a copy of a custom element is created on the same or a different diagram, it can be treated as another instance of the same custom element (the copy changes when the original custom element's source code is modified), or as a new type of custom element (changes to the source code in the original custom element do not affect the copy).

UMLet stores the element's code directly in the diagram (or palette) containing the element, mostly because it eases the exchange of diagrams as they remain single, self-contained files, and don't exhibit additional dependencies on external files. To keep the diagram file as small as possible, only the user's code is stored, not the surrounding template class. Loading a custom element internally works just like generating a new custom element—see section 2.3.1. The obvious drawback is the performance impact when loading a diagram that contains a large number of custom elements. The transparency of the source code-based file format, though, makes up for the performance penalty.

Finally, UMLet treats copies of custom elements as new element types, instead of new instances of the same element type, in order to handle copies of elements within a diagram and between diagrams consistently.

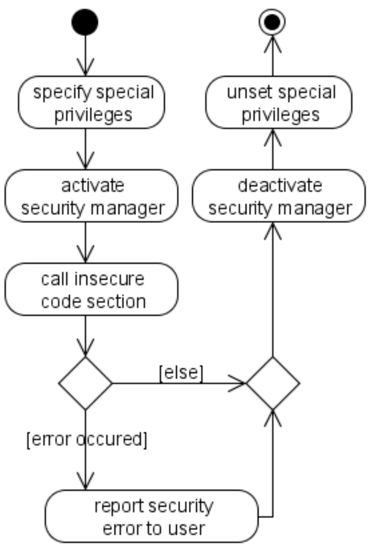

specify special privileges
activate security manager
call insecure code section
[else]
[error occured]
report security error to user
deactivate security manager
unset special privileges

Figure 2.7.: Security workflow

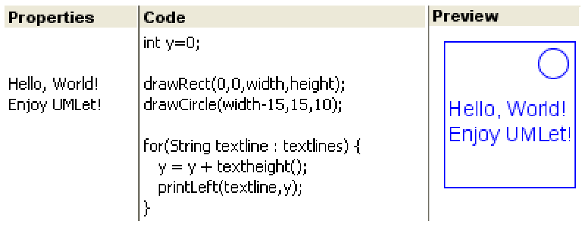
One drawback of this “hidden” element exchange is that an attacker could distribute diagrams containing malicious custom elements. This security aspect will be discussed in more detail in section 2.3.6.

2.3.6. Security

As mentioned in section 2.3.5, loading a custom element may be a security risk if the user is allowed to use the entire scope of the underlying programming language (e.g., a custom element could access the file system). To protect end-user systems from damage, several measures can be taken.

The first measure is very easy to implement and thus commonly used. Whenever a diagram containing custom elements is loaded, a warning message that informs the user about the potential risk is displayed. The diagram is only loaded after the user confirms that the source can be trusted. One problem with this simple solution is that a lot of users will ignore these warnings as they are not aware of security risks or have become insensitive to security related alerts. Even if the user is aware of the risks he may be still be fooled to open a malicious diagram—e.g., if the attacker resembles a trusted source. Another problem is that users may create malicious elements by accident and thus harm their own system during custom element development.

The second security measure does not depend on the user but relies on the programming language’s security mechanism. First, all but the paint method of the abstract custom element



Properties:

```
Hello, World!
Enjoy UMLet!
```

Code:

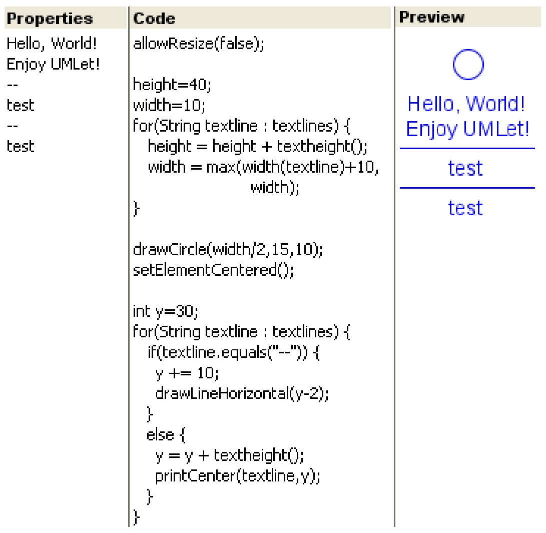
```
int y=0;

drawRect(0,0,width,height);
drawCircle(width-15,15,10);

for(String textline : textlines) {
  y = y + textheight();
  printLeft(textline,y);
}
```

Figure 2.8.: Custom element example 1

class are declared as *final* methods—this ensures that these methods cannot be overridden by the custom element. As a consequence, custom code can only be placed in the paint method, even if the attacker finds a way to break out of it. A security manager (like the one in Java) then enforces that no security critical operations (e.g., file access) are performed during execution of the paint method. To ensure that end users are unable to deactivate the security manager, a random token only visible to the method calling the paint method is used for authentication.

Figure 2.7 gives an overview how a method call to *paint* is handled.

2.4. Examples



Properties:

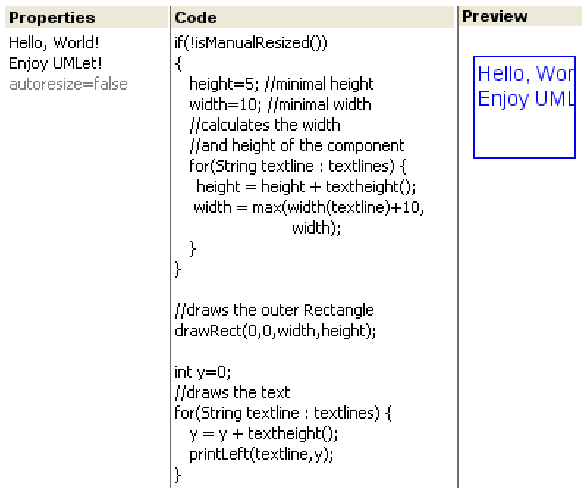
```
Hello, World!
Enjoy UMLet!
--
test
--
test
```

Code:

```
allowResize(false);

height=40;
width=10;
for(String textline : textlines) {
   height = height + textheight();
   width = max(width(textline)+10,
                  width);
}

drawCircle(width/2,15,10);
setElementCentered();

int y=30;
for(String textline : textlines) {
   if(textline.equals("--")) {
     y += 10;
     drawLineHorizontal(y-2);
   }
   else {
     y = y + textheight();
     printCenter(textline,y);
   }
}
```

Figure 2.9.: Custom element example 2

The following four custom element examples give a quick overview of UMLet's live-compilation possibilities. The examples' figures show the property panel, the code panel, and the preview



```
Hello, World!
Enjoy UMLet!
autoresize=false
```

```
if(!isManualResized())
{
   height=5; //minimal height
   width=10; //minimal width
   //calculates the width
   //and height of the component
   for(String textline : textlines) {
    height = height + textheight();
    width = max(width(textline)+10,
                   width);
   }
}

//draws the outer Rectangle
drawRect(0,0,width,height);

int y=0;
//draws the text
for(String textline : textlines) {
   y = y + textheight();
   printLeft(textline,y);
}
```

Figure 2.10.: Custom element example 3

panel.

The *textlines* variable inside the code panel contains the property panel's text. The user code then parses this text to generate the appropriate graphical representation for the preview panel.

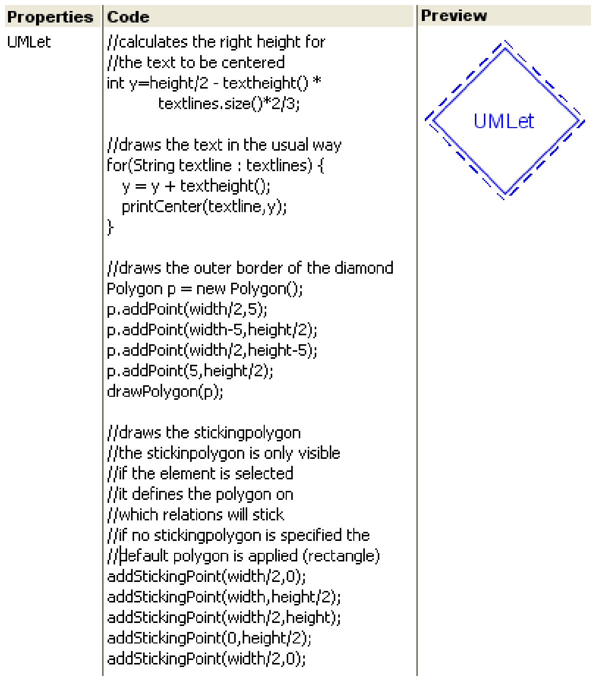
As soon as the source code is modified in the source code panel, the code is recompiled. If it compiles successfully, the code is applied to the text in the property panel and the element's preview is updated. If the property panel's text is changed, the last successfully compiled code version is applied to the changed property text, and the element's preview is refreshed.

The example shown in figure 2.8 creates a rectangular textual element with a small circle in the upper right corner. The *drawCircle* and *drawRect* method calls draw the bounding rectangle and the circle in the element's upper right corner. Finally, the for-loop draws the text that has been entered in the property panel.

Figure 2.9 shows an example with more complex functionality. It prevents the user from manually resizing the element by calling the *allowResize(false)* method and goes on to calculate the component's dimensions. The *setElementCentered* method moves the element's anchor point to its horizontal center. Any update to the element's width is then applied in equal measure on both sides. The for-loop not only draws the text but also tests for lines containing "– –" and draws a horizontal line instead.

The example in figure 2.10 adds optional manual resizing to the element. The *isManualResized* function automatically checks if the element has been resized manually. As soon as the user resizes the element, the *autoresize=false* string is appended to the property panel. The function *isManualResized* returns *true* whenever the property panel contains that string (*autoresize=false*).

In example 2.10, the user has already resized the element manually so the if-block that calculates the size automatically is not executed. As soon as the *autoresize=false* string is removed from the property panel, the size is computed and adjusted automatically again.



```
//calculates the right height for
//the text to be centered
int y=height/2 - textheight() *
        textlines.size()*2/3;

//draws the text in the usual way
for(String textline : textlines) {
   y = y + textheight();
   printCenter(textline,y);
}

//draws the outer border of the diamond
Polygon p = new Polygon();
p.addPoint(width/2,5);
p.addPoint(width-5,height/2);
p.addPoint(width/2,height-5);
p.addPoint(5,height/2);
drawPolygon(p);

//draws the stickingpolygon
//the stickinpolygon is only visible
//if the element is selected
//it defines the polygon on
//which relations will stick
//if no stickingpolygon is specified the
//default polygon is applied (rectangle)
addStickingPoint(width/2,0);
addStickingPoint(width,height/2);
addStickingPoint(width/2,height);
addStickingPoint(0,height/2);
addStickingPoint(width/2,0);
```

Figure 2.11.: Custom element example 4

The last example in figure 2.11 demonstrates the implementation of sticking polygons, which specify where relations stick to a UML element on the diagram. The *addStickingPoint* method adds points to the sticking polygon. At least two points are required. The sticking polygon is drawn as dashed lines. If no polygon is specified, the relations will stick to the rectangle defined by the outer borders of the component. The other methods used in example 2.11 are simple: first the text is drawn at the center of the element; then a diamond is drawn around it.

2.5. Conclusion

Letting users extend applications by programming has been a research topic for quite some time without producing many widely-used results (except custom mathematical expressions). In this chapter we describe a UML tool that enables users to create new graphical UML elements by using the internal graphical API of the tool itself. While this is a very special application, the concept may be applied to other areas.

The approach is simple and provides immediate feedback to the end, addressing Ruyter's requirements for successful EUD applications. A crucial differences to other end-user development approaches is that the development environment is integrated into the main application window and thus no media break occurs. In addition, this chapter addresses several problems including persisting the code, exchanging elements and security.

Future research will focus on an easier exchange of custom elements between end users, potentially over a Web service, and on custom UML relation types.

3. Fast UML Diagram Sketching with Diagrams by Text

Most UML tools follow the easy-to-learn/easy-to-use paradigm and try to allow consumers to use most of the tool's features without ever reading the documentation. Many tools try to accomplish this by making features more accessible via graphical user interfaces rather than cryptic keyboard commands. Although implementing an easy-to-learn UI is not a silver bullet for all usability problems it usually improves the initial user experience while reducing performance for advanced users [27] (see figure 3.1).

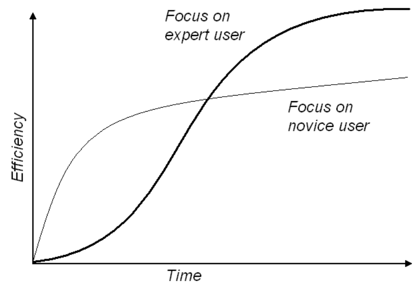


Figure 3.1.: Learning curves - focus on novice/expert

One technique in state of the art software is to introduce keyboard shortcuts for frequent tasks. Unfortunately keyboard shortcuts are usually no help during more complex tasks like adding a special element to a diagram.

UMLet tries to go one step further by providing a method to create entire diagrams using only textual input. The text entered by the user is transformed into a finished diagram. No further actions have to be taken by the user, which saves time and might actually even outperform the pen and paper approach. The simple example in figure 3.2 illustrates how an activity diagram could be generated.

Listing 3.1: Activity diagram - user entered text

```
Start
Activity1
Activity2
End
```

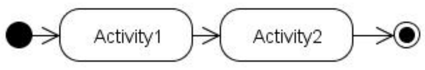


Figure 3.2.: Generated activity diagram

Currently, both sequence and activity diagrams are supported. To ensure a simple initial user experience, the usual drag and drop approach is still available. The performance of both textual diagram input and manual drag and drop diagrams input is discussed in [1]. This paper's results show that UMLet offers great performance benefits when textual diagram input is used. However, these advantages don't come without disadvantages. The major drawback is that a language that is capable of expressing an entire diagram requires additional learning effort.

Section 3.1 introduces the goals of this chapter and section 3.2 presents some related work. The basic approach of a language that expresses activity diagrams will be presented in section 3.3. This approach's drawbacks will be presented in more detail in section 3.3.4. In order to reduce the required learning effort, section 3.4 introduces a hybrid approach that provides both a stripped down text based language for definition of the general diagram components and layouts as well as a mouse based drag and drop logic for positioning and resizing.

Both approaches will be compared to the usual drag and drop functionality within UMLet. For this purpose we make use of a technique presented in [33] and focus on measuring the keystrokes and mouse clicks while ignoring other aspects like eye movement.

3.1. Goals and Approach

The main goals of this chapter are:

- (G4) Provide a way to syntactically define activity diagrams, while avoiding interactive dialogs.
- (G5) Provide a theoretical approach to spread the diagram by text generation approach to other UML diagram types.

- (G6) Quantitatively evaluate the efficiency of this diagram creation process with a performance evaluation based on Tamir's GUI metrics.

Generating UML diagrams using a designated language is very similar to the design of a *Domain Specific Language* [26]. Instead of implementing functionality via executable code, this *DSL* defines a graphical diagram. Thus the user is able to "program" his diagrams.

In section 3.3 we will provide a basic approach that enables the end user to syntactically define activity diagrams (G1). This approach is very similar to the already existing sequence diagram generation technique of UMLet.

Section 3.4 will give a theoretical overview about an hybrid approach which enables the user to change several properties of a generated diagram (e.g. position of elements) after it has been generated from the entered text. This enables the user to position elements at will, which is necessary to spread the approach to other diagram types (G2).

Finally the performance of the different approaches will be evaluated using Tamir's GUI metrics [33] (G3). This will highlight the strengths and weaknesses of the different approaches.

3.2. Related Work

Sketching UML diagrams is a wide spread topic. Doing this using a textual grammar is a rather rare one. Most of the literature regarding text based diagram generation focuses on generating UML diagrams from existing code—reverse engineering [36].

Other approaches [22] propose the use of speech recognition to avoid in depth dialogs. Although these approaches share major goals (e.g. increase efficiency) with the text based approach the main difference is that the speech recognition approach specializes on making the access to the elements detailed properties easier, while the text based approach specializes on generating a basic diagram as fast as possible.

The current work on declarative diagram generation [31] focuses on creating a diagram from a given text. The approach focuses on reducing the complexity of the text that is necessary to draw a diagram—compared to source code. The difference to our approach is that most tools (e.g. UMLGraph[1], MetaUML[2]) that support this approach still have a media break. The tools usually consist of an executable file that accepts one input file containing the text and then produces one output file containing the diagram. A tool that already avoids this media break

[1]`http://www.umlgraph.org/doc.html`
[2]`http://metauml.sourceforge.net/statemachine-diagram.html`

(SEQUENCE[3]) just supports sequence diagrams.

3.3. Basic Approach

A previous UMLet paper [1] already discusses the generation of sequence diagrams. We therefore focus on generating activity diagrams even though the method may be adapted for other diagram types. Activity diagrams are—similar to sequence diagrams—relatively easy to implement as a textual language because their layout is deterministic. Compared to the positioning of elements in class diagrams where good layouts are difficult, the layouts of activity diagrams are usually simple.

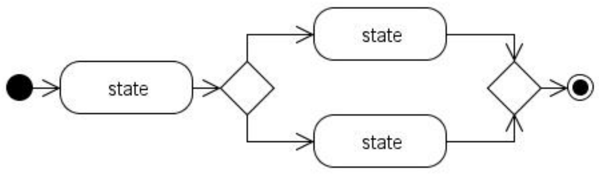


Figure 3.3.: Typical activity diagram

As shown in figure 3.3, most activity diagrams are drawn in a specific direction, either left to right or top to bottom, to represent a workflow. This observation is used to position the elements within the diagram. The detailed layout generation is discussed in detail in section 3.3.3.

To generate the visual representation of the user entered text, the task is split into two parts:

1. Generate a model from the textual representation
2. Generate a visual representation from the model

One of the advantages of using an intermediate model [23] is, that a model representation makes the approach more efficient, flexible and simple [7]. The visual and textual representations are loosely coupled. Thus if the grammar changes, no changes to the graphics generation engine are required and vice versa. The application's flexibility increases because the model representation may be used to translate the diagram to any other format. Thus code generation or other visual representations may be added independent of the main target format.

[3]`http://www.zanthan.com/itymbi/archives/cat_sequence.html`

3.3.1. Model Definition

The following UML model describes the data that is used as separate level between the textual and visual representation. The model represents the general nature of activity diagrams. After the model is generated out of the text, it may be transformed into almost every other form. In this case, a visual representation is sufficient, but an export to specific file types, or even code generation are possible.

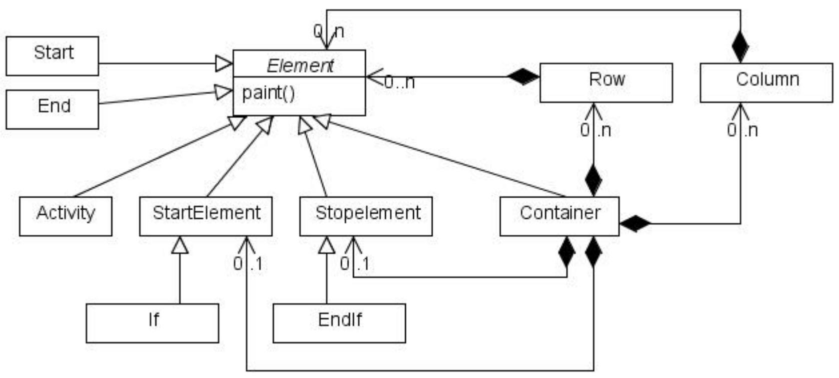


Figure 3.4.: Activity diagram model definition

The model shown in figure 3.4 has one central component. Every element is a subclass of the *Element* class. This ensures that all components of an activity diagram can be inserted at any position. In general, there are six subtypes of elements:

1. Normal elements (like the activity element) with incoming and outgoing transitions
2. Elements with outgoing transitions (Start)
3. Elements with incoming transitions (End)
4. Elements that may be a container's start element (StartElement)
5. Elements that may be a container's end element (StopElement)
6. Elements containing other elements (Container)

The most complex element is the container element. It is used when one component has more than one outgoing transition. It contains a grid (specified by rows and columns) that includes the elements that are specified within the container. It may also include a starting element and/or ending element that is inserted at the start or end of the container respectively. This element is then used for splitting the workflow into multiple parallel or exclusive activities.

Transitions are not stored within the model and are therefore saved implicitly with the position and type of the specific element. Every element that permits outgoing transitions connects to every successor that permits incoming transitions. This technique also simplifies the textual representation of the model because no transitions have to be specified by the user. One disadvantage of this approach is that the user is unable to connect arbitrary elements. To overcome this problem the special *->element*(see table 3.2) is used to insert transitions between arbitrary elements.

3.3.2. Text to Model

The textual representation's goal is to provide both an easy-to-use, easy-to-learn language to generate activity diagrams and access to most of the commonly used UML activity diagram elements. To keep it simple, there is only support for elements that do not contain other elements - like UML's partitioning feature.

The first task was to describe serial activities of an activity diagram. Serial activities can be modelled by putting each activity on a single line of the text. Thus the following lines result in the activity diagram specified in figure 3.5.

Listing 3.2: Activity grammar example 1

```
Start
Activity1
Activity2
End
```

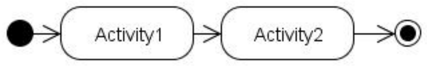


Figure 3.5.: Activity text grammar example 1

An activity diagram can also consist of elements other than activities. A short summary of all available elements within the current version of the grammar is shown in table 3.1 and table 3.2. The elements are used like activities and generally provide the same behavior while

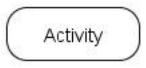
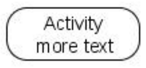
Text (in grammar)	Incoming/ Outgoing Transitions	Activity diagram Element
Start	Outgoing	●
End	Incoming	◉
AEnd	Incoming	⊗
Activity	Both	
Acticity \ more text	Both	
If	Both	◇
EndIf	Both	◇
Sync	Both	▬
Fork	Both	▬
[condition]	Both	

Table 3.1.: Basic textual elements of the activity diagram language

having different graphical representations. Another difference is that other elements may not have both incoming and outgoing transitions to adjacent elements.

The next step is to model parallel activities in addition to serial activities as shown in figure 3.3. To achieve this we introduce the possibility to generate containers of elements via tabs. Every element is categorized according to the number of tabs at the beginning of its line. This tab count is referred to as *tab index*. When the tab index increases, a new container is generated within the current container. To add a new column to a container, an empty line has to be placed between two blocks having the same tab index. When the tab index decreases, the current container is closed and subsequent elements are added to the parent container. The

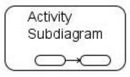
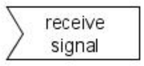
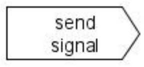
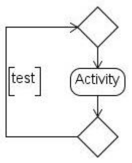
Text (in grammar)	Incoming/ Outgoing Transitions	Activity diagram Element
Activity\Subdiagram..	Both	
>receive\signal	Outgoing	
send\signal>	Incoming	
|	Both	—
While[condition] Activity	Both	
->element	special	This is a element that connects its predecessor to the specified element (therefore cancels the usual connection—if any—to the successor of the element).

Table 3.2.: Advanced textual elements of the activity diagram language

following example code generates the activity diagram shown in figure 3.3.

Listing 3.3: Typical activity diagram - textual representation

```
Start
state
If
  state

  state
EndIf
End
```

Generally, *If* or *Fork* commands should be used before a parallel activity is started because the UML standard does not allow activities with more than one successor. Therefore we included a feature that adds an *If/EndIf* element if no *If/EndIf Sync/Fork* elements are provided at the start or end of the container. Thus the following code is equivalent to the code above.

Listing 3.4: Typical activity diagram - alternative textual representation

```
Start
state
  state

  state
End
```

Synchronization bars can be added to the diagram via *Sync/Fork* keywords. The following code provides more details for the containment mechanism and creates the diagram shown in figure 3.6.

Listing 3.5: Activity grammar example 2

```
Start
state1
Fork
  state2
  state3

  state4
    [condition1]
    state5

    [condition2]
    state6
Sync
End
```

3.3.3. Model to Visualization

After the model is created it is translated into the graphical representations that were shown in the examples in section 3.3.2. Every element has it's own paint() method where the position and size of the element are calculated before the element is drawn to avoid unnecessary multiple computations. Then the transitions are inserted which is trivial since all element positions are known.

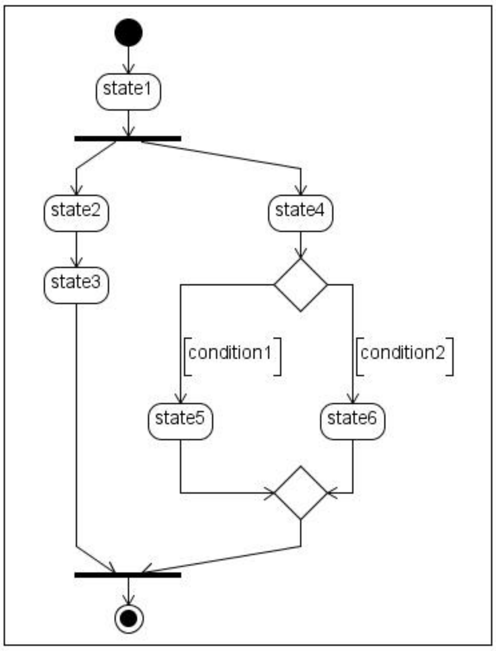


Figure 3.6.: Activity text grammar example 2

3.3.4. Drawbacks

Although diagram generation by text achieves a great performance increase (see section 3.5 for details) it does not come without drawbacks. The main drawbacks of this technique are:

1. The user has to learn a new domain specific language to generate diagrams
2. The flexibility of diagram generation is restricted by the limitation of the underlying language.
3. This approach may not fit other types of diagrams (e.g. class diagram) where element positioning is a non trivial task.

The first disadvantage results from the assumption that creating a diagram by drag and drop is much easier to learn than to create it with a domain specific language—even if good examples are provided. It would be beneficial to merge both variants into one that helps the user to learn the domain specific language by creating his own examples via drag and drop. After a while the user might have understood the generated text and starts to use the more effective text base approach.

The second drawback mainly arises because it is very difficult to integrate element positioning in the textual approach. This would require a change in the generation engine (e.g. activity diagram partitioning).

The third disadvantage is critical because it limits this approach to specific diagram types. Other diagram types like class or use case diagrams might be used more often than activity or sequence diagrams. Within class or use case diagrams, automatic positioning of elements is a non-trivial task because an element's position is usually strongly tied to its structure and importance, which might not be visible to the tool. Although the literature [12] provides several algorithms to paint such diagrams in a well formed way, the main problem becomes that the diagram would change vastly during live feedback. This confusing behavior would continuously disrupt the users efforts to create a specific layout. Thus the only possibility to create a language for these diagrams is to allow for screen coordinates within the text. Another problem of class diagrams is the vast amount of information that may be added to a single class element (e.g. methods, attributes). If all this information for all elements is added to a single text field, the textual representation of a diagram grows to an infeasible size.

To overcome this drawback, a hybrid approach might be the solution. The basic diagram structure can be created using the language and then customized using regular techniques.

3.4. Hybrid Approach

To address the drawbacks discussed in section 3.3.4 we will now introduce a hybrid technique that involves both diagram generation by text and by mouse. However, we will not introduce this approach for activity or sequence diagrams because apart from the increased learning effort activity and sequence diagram generation by text seems to work well. We will create a text-based approach for class diagrams that allows for standard mouse routines where applicable. It can be easily adapted to support activity and sequence diagrams which is discussed in section 3.4.8.

The goals for this approach are as follows:

1. Provide an easy to use language to generate simple class diagrams
2. Enable the user to choose between diagram generation by text and by mouse
3. Alteration by the mouse shall affect the text, and vice versa

4. Resizing and placement of elements is done by mouse, except for initial placement of text generated elements
5. The mouse diagram generation interactions must not differ from the standard diagram sketching process
6. All elements can be added to the diagram, including elements that are not in the textual grammar

The main benefit is that users are able to start diagram sketching as usual while generating the diagram's textual representation. Over time, users learn to create diagrams by text. This is a process that is frequently referred to as *active learning* [25] or learning by doing. Every single diagram becomes a type of tutorial for the text generation language and therefore only requires minimal learning effort (drawback (1)). Another goal is that any element may be added to a diagram including those that don't exist in the underlying text based language (drawback (2)). To overcome drawback (3), manual positioning and resizing by mouse is utilized.

To achieve these goals, it is necessary to link basic diagram features to the textual language and vice versa. Thus the underlying model has to be bidirectional to keep both the graphical representation and the underlying text synchronous. This is accomplished through several mappings between the various models that have to be implemented [11]:

- model to model transformations
- model to text transformations
- text to model transformations

Additionally the model has to have a graphical representation to be shown on the diagrams panel.

We will define a basic model and different mapping processes between text, model, and visual representation in the following sections. However, we will not provide a specific language to implement. A summary of some different model transformation languages is listed in [18].

3.4.1. Model Definition

Defining a model for class diagrams is a lot easier than defining a model for activity diagrams as transitions and positioning information does not have to be stored implicitly within the model. The model in figure 3.7 includes a list of class elements, transitions and other elements which may be connected to each other. Some UML limitations like the limitation that class elements cannot be connected to use case elements are ignored, as it would unnecessarily restrict the user without gaining any benefits.

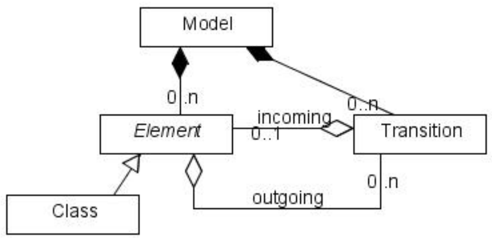


Figure 3.7.: Model for class diagrams

It is necessary to maintain a list of transitions because transitions are not necessarily connected to elements. Another prerequisite is to track unknown elements because it is possible that non-class elements are included in a class diagram. The position is stored with each element and may only be altered by dragging the element to another position by hand. The initial placement of elements that are created via text input will be discussed in section 3.4.2.

The model in figure 3.7 does not fit the diagram's textual representation because the grammar only contains class elements and transitions between class elements. Transitions that don't connect to exactly two class elements are not included in the model in figure 3.8.

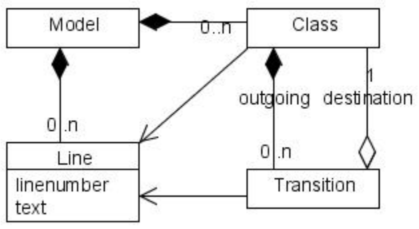


Figure 3.8.: Textual model for class diagrams

This altered model shows the basic textual grammar presented in section 3.4.2. Since it differs from the model specified in figure 3.7, a bidirectional mapping between the two models is required. This mapping is described in section 3.4.4.

3.4.2. Text to Model

The textual representation is specified by the following EBNF [19]:

Listing 3.6: Class diagram EBNF

```
Text = [Lines];
Lines = Line ["\n" Lines];
Line = Classname ["~" ClassId] [Connections];
Connections = Connection [Connections];
Connection = ["[" Multiplicity "]"] Arrow ["[" Multiplicity "]"]
     ClassReference;
ClassReference = Classname | ClassId
```

The above EBNF lacks the definition of three nonterminal symbols: *Classname*, *ClassId*, *Arrow*, and *Multiplicity*. These definitions are intentionally omitted as they depend on the respective requirements for naming or arrow conventions. Basically a class is defined by putting its name and id onto a single line. Following the class definition, an arbitrary amount of connections may be added to the element, all on the same line of text. The connection end points have to be valid class ids. If no class id is specified, the id is set to the class name. If two elements have the same id, the connection is connected to the first one found. The multiplicity of a transition is included in the grammar because it is used often, even in very simple diagrams.

The textual representation can be directly mapped to the model defined in figure 3.8. First, every line in the model is updated with the corresponding text line. Then the class and transition information is extracted. One necessary measure is to store the model's lines in the same order as the text's lines to ensure that the user defined order is maintained.

The most critical problem is updating an existing element from text. To identify the corresponding element, the altered line is checked for a referencing class or transition element. Depending on the type of change, the corresponding element is then updated rather than generated to ensure that additional information (e.g. methods, attributes) is not lost during class name or association updates.

A problem occurs as soon as a line is cut and reinserted at a different position. This causes the loss of all special information including positions, functions, methods, etc. To prevent the user from deleting this information by accident copy/paste functionality should be forbidden. Therefore this problem only occurs if undo/redo operations are applied to the grammar. To avoid the problem the undo/redo mechanism has to be altered in a way that does not merge multiple state changes of diagrams.

Another issue during the mapping process is saving the text's invalid data (text that could not be transfered to the model and is thus only persistent within the text). This prevents information loss when a line of text is unfinished and enables the user to insert comments. The data is stored in respective line objects in the model. A line object represents one line and classes and transitions maintain a reference to their corresponding line. This ensures that data that does not fit into the model is not lost.

3.4.3. Model to Text

Whenever the model is altered, the whole text has to be regenerated. Mapping classes and connections is simple because line objects are automatically updated to reflect changes. New classes are added to the end of the class list so the class order stays the same. As soon as the model is consistent, the line objects emit text directly to the text field.

When a class is deleted all references to it are also removed. As a consequence all incoming and outgoing arrows are also removed from the model. The lines in the textual representation are still visible, but marked as invalid and thus not translated back to class or transition objects.

3.4.4. Model to Model

The model transformations between the textual (figure 3.8) and visual models (figure 3.7) are simple. A visualization of the mapping between the two models is shown in figure 3.9. The major rule applied to all mappings is that information within the model elements are not deleted unless the entire model is deleted. For example, a class's function names are not removed if the textual model does not contain the function names. Another important aspect is that the mapped classes store their corresponding counterparts after a mapping has been applied once. This ensures that the additional information that is available in both models is not lost, because a model element is recreated instead of updated.

In order to map the **textual** to the **graphical** model the textual model's class information has to replace the visual model's class information. Therefore items that exist in the visual model and do not exist in the textual model are deleted. In the opposite case the class is inserted. Every transition of the textual model is mapped to a transition in the visual model and every outgoing connection of a class to another class is mapped to the equivalent in the visual model. A transition's destination connection is transformed to an element's incoming connection. If a new class or transition is added to the graphical model it is assigned a default coordinate and default size that can depend on the specific situation. Classes and transitions are only deleted

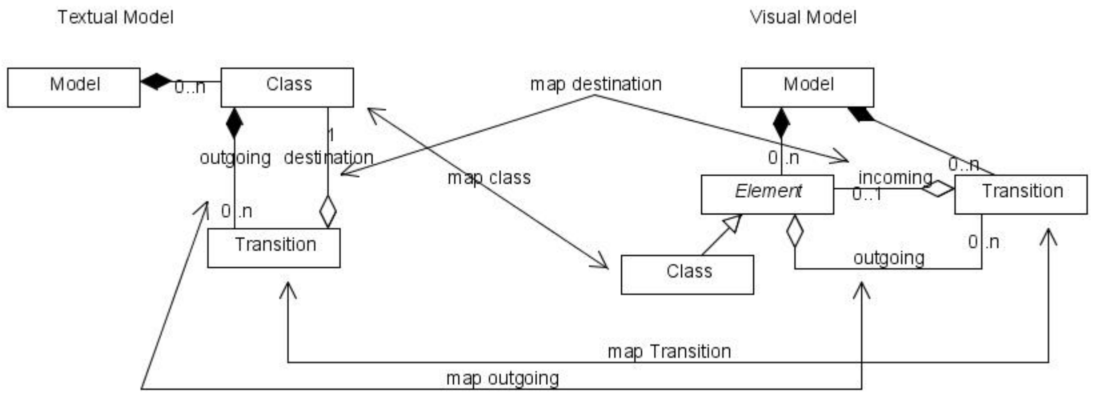


Figure 3.9.: Mapping between textual and visual model

from the graphical model if they are present in both models.

The process of mapping the **graphical** to the **textual** model is mainly the same. All classes of one model are mapped to a class in the other model. The transition mapping is a little bit more complex since only a subset of transitions is transferred to the textual model. In fact, only transitions that connect two classes are mapped to the textual model. Transitions that lead to no class, or connect to an element other than a class are ignored.

3.4.5. Model to Visualization

The model can be visualized by painting the components one by one. Because the elements' positions and sizes are stored within the model all components can be drawn without additional computing.

3.4.6. Visualization to Model

Removals and updates of elements that occur in the graphical user interface are simply translated to the visual model. The only exception is the insertion of new elements or transitions. When a new element is inserted, its coordinates and dimensions have to be set to default values. Inserting transitions is more complicated because incoming and outgoing ends need to be detected. The solution can be simple. Incoming and outgoing ends will be chosen arbitrarily but consistently, i.e. the mapping is always done the same way. The user must not be confronted with changes at a point of the model that he did not change.

3.4.7. Choosing Diagram Type

An additional difficulty is the feature that allows any UML element to be used in any diagram, which requires a mechanism to detect which mapping (e.g. class or activity diagram) should be applied. Only one mapping can be applied to one diagram sheet at one time. Otherwise the text may become inconsistent as the different grammars may be overlapping.

One alternative is letting the user specify the type of the diagram in advance. This would give the user full control over the diagram type but would differ from UMLet's current behavior. Thus users might not be aware of the possibility to add e.g. use case elements to a class diagram.

To avoid this inconsistency, the diagram type is finalized when the first element is added to the diagram. Users that are used to the current version of UMLet will not experience a change in the workflow—goal (5). One drawback may be that the user does no longer control the diagram type. Consequently, if the user first adds a use case element to a sheet that she intends to use as a class diagram, it would in fact become a use case diagram.

Fortunately, allowing the user to manually change the diagram type can solve this problem. This solution also has another beneficial effect. Sometimes it is necessary for two diagram types to be on the same diagram sheet. By changing the diagram type of the sheet after one diagram has been finished, the second diagram may again be created using another textual grammar.

3.4.8. Adoption to Activity Diagrams

The class diagram approach may be easily modified to support other diagram types that do not store the position of the elements within the textual representation (e.g. usecase diagrams). Supporting activity diagrams however is a little bit more complicated. The main problem is that the current activity diagram approach stores the positioning information implicitly within the text. Therefore, if an element is moved, the automatic positioning information is lost which hinders position updates of inserted or removed elements.

To overcome this issue, the tool has to provide an option to reinstate automatic positioning for elements. This can be implemented in two ways:

1. Provide a graphical user interface option to switch back to automatic mode (e.g. via double click or the context menu)
2. Persist the manual assigned coordinates to the text (e.g. [12,123])

Both options solve the problem but the first one is easier to understand, especially when a transition is assigned more than one coordinate. The other approach would render the transition as "[0,0]-[10,10]-[10,50]->[50,50]" as opposed to "->".

Another problem is that transitions can be moved to invalid states (e.g. not connected to an element, or an element with two outgoing transitions...). In the case where an element that should have incoming transitions but doesn't have any because they were removed is regenerated, an additional transition would be added to the diagram. To avoid redundant transition generation we introduce the new special element "-" that simply cuts a transition before or after an element. This is demonstrated in the following two listings and figure 3.10.

Listing 3.7: Activity diagram - without cut element

```
Start
Activity1
Activity2
End
```

Listing 3.8: Activity diagram - with cut element

```
Start
Activity1
-
Activity2
End
```

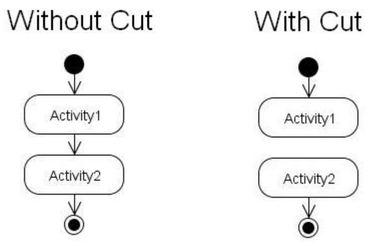


Figure 3.10.: Activity diagram - cut element visualization

After considering these two special cases, the remaining parts can easily be mapped from the class diagram to the activity diagram approach even though it may still lead to unwanted diagram modifications when an element is deleted. In this case, an entire if-block might be removed as a consequence of the changes in the textual representation.

3.5. Performance Analysis

We will now compare the performance gap between the three different activity diagram approaches and the two different class diagram approaches presented in the sections before. Previous performance analysis results [1] indicate that the performance increase of the textual grammar compared to the usual drag and drop method is significant.

The previous paper's [1] use cases are insufficient for this analysis because they mainly tested the basic class diagram features and merely compared sequence and class diagrams. Therefore we define new use cases in section 3.5.1 which will cover both simple and complex class diagrams and activity diagrams. We will compare the creation of an entire diagram rather than comparing isolated modifications because the latter are very different to handle and sometimes unavailable in certain approaches.

The results will be presented in sections 3.5.4 and 3.5.5. The precise steps taken to deduce these results are documented in appendix A.

3.5.1. Definition of Use Cases

This section provides four diagrams, two class diagrams and two activity diagrams. Each one of these diagrams represents a use case that has to be solved using the presented techniques. All diagrams were designed to have possible solutions in any of the approaches.

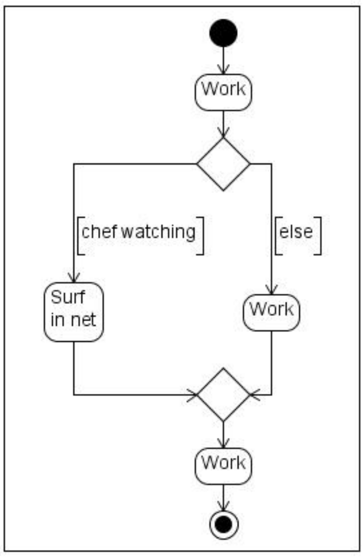


Figure 3.11.: Use case: activity (simple)

One general condition is that the diagram needs not to be drawn in the exact layout provided here. Any layout is sufficient as long as the meaning and general structure of the diagram is clear—e.g. one element has to be below another and has to be connected to it, but it does not need to be exactly three pixels below it.

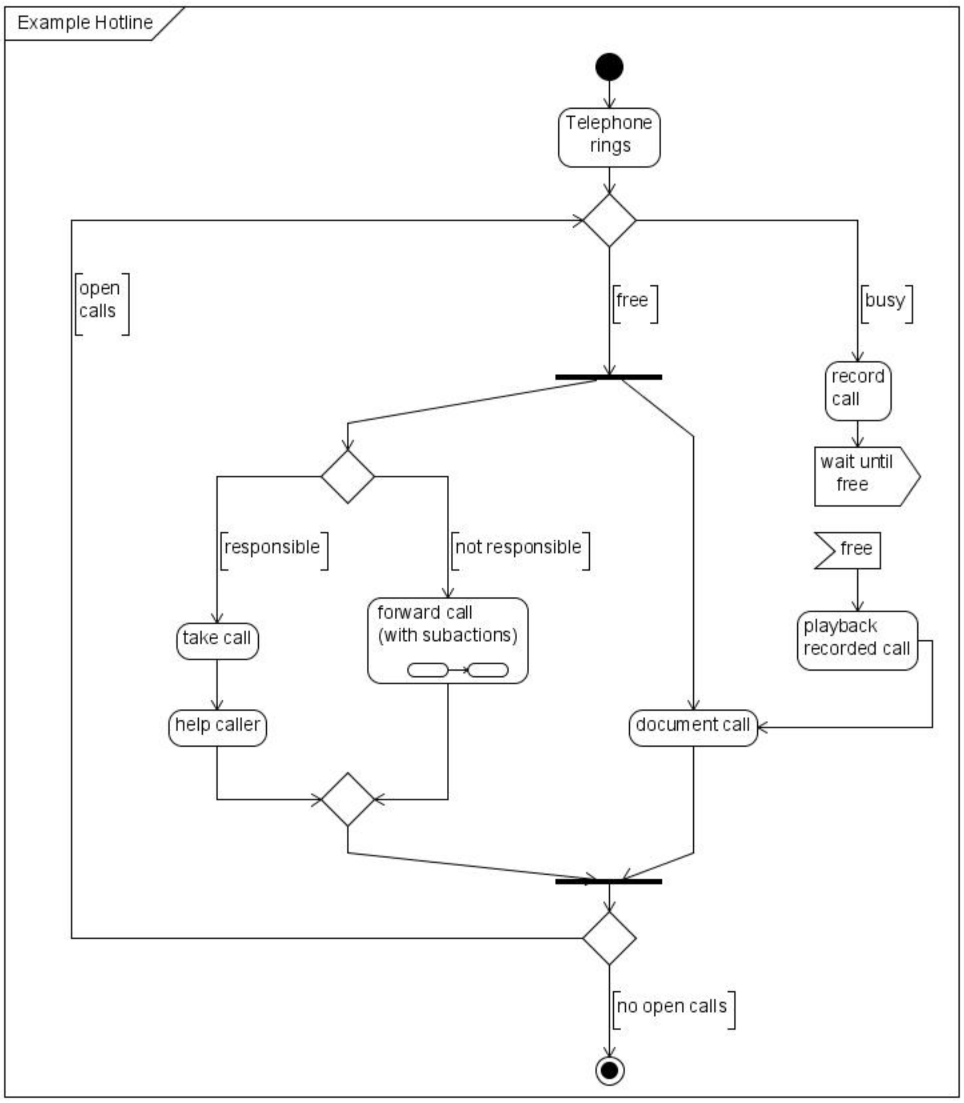


Figure 3.12.: Use case: activity (complex)

The simple activity diagram use case in figure 3.11 only consists of some activities, conditions, and two *If* elements. The complex activity diagram however—figure 3.12—makes use of the entire set of possible elements provided by the activity diagram's textual grammar.

The diagram documents a typical telephone hotline workflow. More complex components, like diagram partitioning, are not included here.

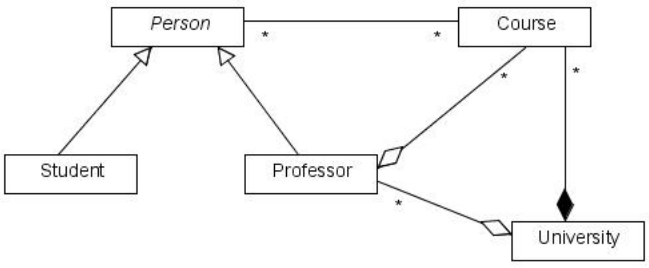


Figure 3.13.: Use case: class (simple)

The simple class diagram shown in figure 3.13 is a sketch of classes without detailed information like methods, variables, or transition descriptions. It models a typical university that provides courses for students.

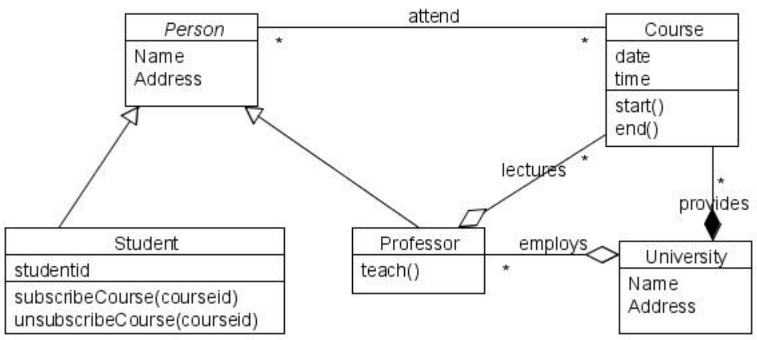


Figure 3.14.: Use case: class (complex)

The second class diagram visualized in figure 3.14 is an exact copy of the first one with the difference that each class specifies its variables and methods. The first example diagram can be created using textual grammar where only the element positioning has to be done manually. When creating the more complex diagram, the user has to manually add the method and function names to the elements.

3.5.2. Definition of Method

According to the procedure presented by D. Tamir, O. V. Komogortsev and C. J. Mueller [33] we define the effort required to create a diagram as:

$$E = k_1 * mc + k_2 * ks + k_3 * mm + k_4 * smk$$

where mc, ks, mm, and smk define the amount of mouse-clicks (mc), keystrokes (ks), mouse-movements (mm) and switches from mouse to keyboard and vice versa (smk). Drag & Drop actions are counted as mouse-movements. Double-clicks are tracked as single mouse-clicks and special characters that involve more than one key are counted as two keystrokes. The constants k_1 - k_4 define the weight of the different user actions. The constants are set under the following assumptions.

1. The user types about five letters per second.
2. The user is able to click the mouse two times per second.
3. The user can move the mouse to any position within a second.
4. The user is able to switch from mouse to keyboard (and vice versa) in about 1.5 seconds.

Due to these assumptions the constants are set to:

$$k_1 = 2, k_2 = 1, k_3 = 3, k_4 = 6$$

Since the constant values are taken from a single group's experience, we provide the results for different user actions before calculating the overall effort. This makes it easier to re-evaluate the test using different weighting constants.

After substituting the constants into the formula the overall effort can be calculated as follows:

$$E = 2 * mc + 1 * ks + 3 * mm + 6 * smk$$

For every method, it is assumed that both the mouse and the application are in an ideal state to complete the tasks. Thus the correct palette is selected and the mouse is positioned either on the element that is inserted first or on the property panel.

3.5.3. Definition of Compared Approaches

First the different approaches of creating activity diagrams are compared:

1. drag and drop
2. text only
3. hybrid

Obviously the hybrid approach matches the text only approach because it makes use of exactly the same diagram generation language. To compare them we will assume that the hybrid approach's textual representation does not support connecting non-adjacent elements.

The comparison of the class diagrams is done without further restrictions.

1. drag and drop
2. hybrid

As the hybrid approach is not yet implemented we make assumptions on how the simple diagram would be generated. First the diagram in figure 3.15 is generated via the following text.

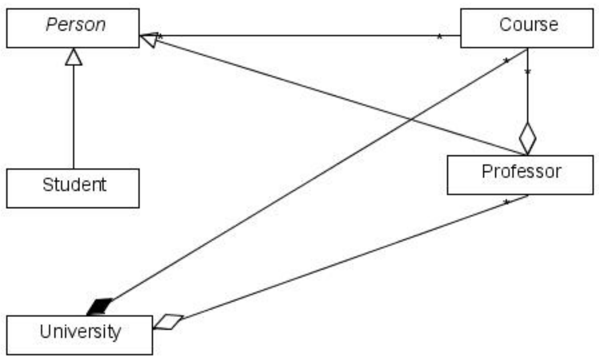


Figure 3.15.: Use case: class (simple) - diagram after generation by text

Listing 3.9: Use case: class (simple) - textual representation

```
/Person/ [*]-[*] Course
Course
Student ->> Person
Professor ->> Person ->>>[*] Course
University ->>>[*] Professor ->>>>[*] Course
```

Then the classes and relations are repositioned using drag and drop operations. The complex use case is basically generated in the same way. After the basic diagram in figure 3.15 has been generated, the different transitions and elements are repositioned. Additionally, detailed information is added to the elements.

3.5.4. Comparison of Activity Diagrams

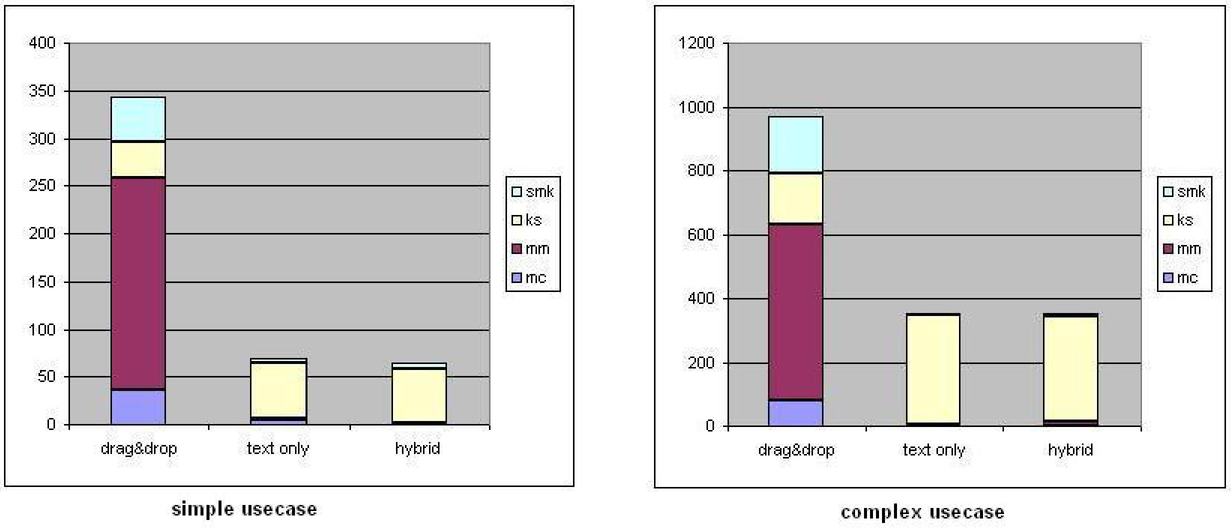
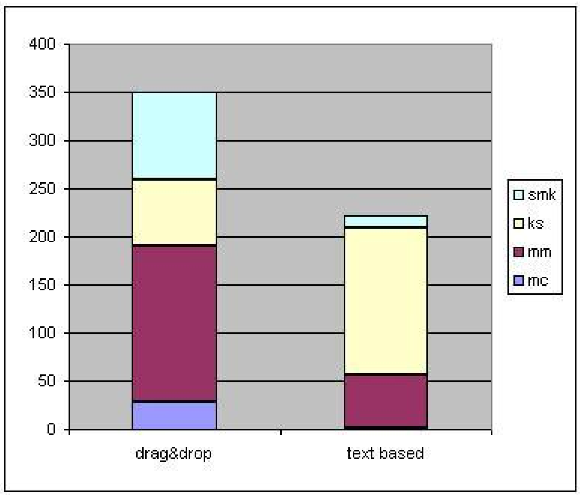


Figure 3.16.: Performance - activity diagram

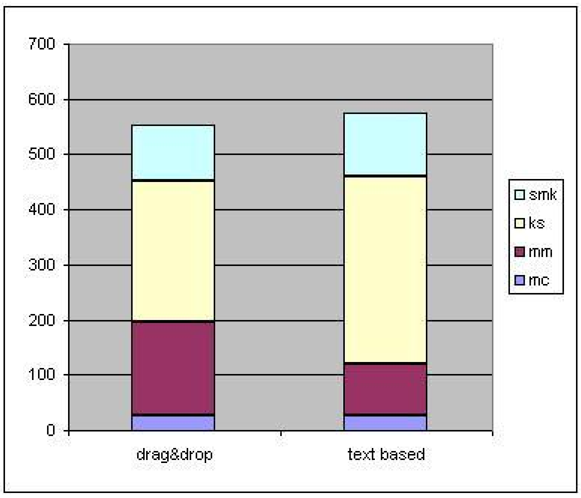
Figure 3.16 compares the different approaches' performance. Since the hybrid approach mimics the text only approach in almost all aspects, these two only differ slightly. Another apparent fact is that the text-based approaches require mostly keyboard input. This is not surprising because the implicit positioning makes manual movement unnecessary.

The figure shows that an activity diagram can be drawn in 20-40 percent of the time when using one of the text base approaches. This difference varies depending on the amount of the text entered into single elements and the additional positioning text that is required for drawing parallel activities. The text affects the percentage because it is added for all approaches and therefore does not yield any performance gain.

The detailed results are shown in appendix A.2.



simple usecase



complex usecase

Figure 3.17.: Performance - class diagram

3.5.5. Comparison of Class Diagrams

Unlike the comparison of activity diagrams, the results for simple and complex use cases are quite different. For the simple use case the text-based approach is—as expected—about 35% faster than the drag and drop approach.

On the other hand both approaches seem to require similar effort for the complex use case , with the drag and drop approach being slightly faster. This is because switches between mouse and keyboard are required in both cases since adding additional information to the classes requires element selection.

If however the user first draws the simple diagram and then updates it to the complex one—which is similar to drawing it with the text based approach—the text based approach again beats the drag and drop, because the actions that have to be performed are the same in both approaches and the text based approach beats the drag and drop approach during generation of the simple use case.

The detailed results are shown in appendix A.1.

3.5.6. Summary of Results

The results of the previous sections indicate that if a good positioning mechanism is provided and all information can be entered into the grammar at once, a great performance increase can be achieved.

If the positioning cannot be implemented in the textual grammar, the performance gain is approximately 50 percent. The worst case is when additional information that cannot be entered through a diagram's textual representation has to be entered directly . This results in no performance gain.

It can be concluded that textual grammars best fit for diagrams that contain elements with all of their information in the grammar. Another beneficial property is that positioning can be done without having meta-information—like the positioning for activity diagrams.

3.6. Conclusion

Within this chapter we introduced an approach for creating activity and class diagrams via text. Because the class diagram positioning cannot be specified in text, a hybrid approach was introduced. The hybrid approach uses a textual grammar to create basic elements and the connections between them while the element positions as well as additional information are added as usual.

The textual approach of activity diagrams performs much better than the usual drag and drop approach—even when drawing more complex diagrams. The hybrid class diagram approach performs best when no additional information (e.g. methods or variable names) is added to the elements. If additional information is added the hybrid approach only performs as good as the drag and drop approach.

It can be concluded that textual grammars are most likely to be used for drawing sketches that do not contain detailed information and best fit diagrams that allow automatic positioning.

4. Embedding User Generated Elements into Textual Grammars

In chapter 2 we introduced an approach to enable the user to create new graphical elements by using an embedded development environment within the tool UMLet. Chapter 3 proposes to generate diagrams using textual grammars.

Textual grammars already support the use of elements that are not designed for the grammar (i.e. don't have a textual representation within the grammar) by adding the elements to the graphical view without displaying it in the textual representation. This ensures that the user generated elements presented in chapter 2 can also be added to text based diagrams, although they cannot be displayed in the textual representation. Thus they cannot be added to a diagram using the text-based approach.

In this chapter we will present a new approach that is based on the one presented in chapter 3. It will enable the user to embed custom elements into a diagram's textual grammar. The approach presented in chapter 2 will be referred to as *original approach*.

4.1. Goals and Approach

The main goal of this chapter is:

- (G7) Provide a theoretical approach to design user customized elements for textual grammars.

In the original approach the user only had to implement the paint method to draw an element. The paint method was subsequently inserted into a predefined template, which was compiled and invoked as a new element.

However, implementing a single method is not sufficient for embedding the element into a textual grammar. To achieve this, several changes to the template engine, the user interface, and the element interface are required:

- The elements have to provide a standardized interface.
- The template has to support all interface methods.
- The user interface has to support template editing.
- The grammar has to be changed to be able to add new elements.

In the original approach the element's interface varied, depending on the specific diagram type's needs. For this approach the standardized interface is preferred since elements should be defined consistently across diagram types. This ensures that the user is not confused with different user interfaces for different diagram types.

The element's interface presented in section 4.2 is based on the activity's textual grammars and class diagrams shown in chapter 3. Additional changes may be required to support additional diagrams.

The template and the user interface then have to support editing all the functions provided from the element interface as well as the assignment to a diagram type. This will be described in more detail in the sections 4.4 and 4.5.

Another important feature is to allow the user to specify a regular expression that differentiates the element from other elements within the textual grammar. Consequently, a mapping of incomplete element definitions to the element state shown in the elements details and vice versa has to be provided. The mapping functions are listed in section 4.3.

To enable the grammars to accept custom elements, they have to provide methods to register new elements. Thus the elements cannot be hard coded within the grammar but need to be loaded as soon as a new element is loaded by a diagram or palette. That means that the elements register themselves with the grammar, which ensures that custom elements can be exchanged by simply exchanging diagrams.

4.2. Element Interface

Providing a simple drawing method like in the original approach is not enough. To fit into the new approach, the element's interface has to provide functions for:

- Painting the element
- Assigning the element to a diagram type

- Deciding if an element accepts incoming and/or outgoing relations
- Default connections to connect relations to
- A regular expression to identify the element
- A mapping function to map the grammar data to the detailed data and vice versa

The paint method has already been used in the original approach. No changes are necessary.

The diagram type is assigned by returning it as a string from a method. This method is represented by a select box instead of letting the user enter code directly. Another solution is to provide separate custom element creation menu items for each diagram type.

The method responsible for deciding whether incoming or outgoing connections are allowed is implemented as check boxes. These two methods are currently only needed for the activity diagram approach because all elements in the class diagram approach allow outgoing and incoming connections. Note that even if no connections are allowed within the grammar they can be created via drag and drop. The only difference is that the element is not mapped to the textual grammar and vice versa.

A more complex method is to provide default connection points for elements. Within this method the user has to provide an algorithm to provide points for each direction (left, right, top, bottom) where relations should be connected. This has to be specified in code as complex operations might be required to express the algorithm.

To define when a custom element is addressed within a textual grammar, a regular expression has to be provided by the end user. As soon as the expression matches a string within the grammar the element will be inserted at the appropriate position. It is important that the elements' expressions do not overlap because it may lead to unexpected results when using the grammar.

The last step is to provide two mapping methods. These are required because the grammar cannot contain all of the element's details (e.g. colors). Therefore one method maps the element representation within the grammar to its equivalent within the element detail and a second method provides the inverse mapping. A more detailed description of these methods as well as an example will be provided in section 4.3.

The following example gives a short overview of the parameters, return values and default values. The abstract functions do not have a standard implementation.

Listing 4.1: Custom elements (textual grammars) - element interface

```
abstract void paint();
abstract String getDiagramType();
boolean acceptIncomingConnections() { return true; }
boolean acceptOutgoingConnections() { return true; }
Point getDefaultCoordinate(Direction dir) {
  if(dir == Direction.TOP)
    return new Point(this.width/2,0);
  else if(dir == Direction.RIGHT)
    return new Point(this.width,this.height/2);
  else if(dir == Direction.BOTTOM)
    return new Point(this.width/2,this.height);
  else
    return new Point(0,this.height/2);
}
abstract String getElementExpression();
abstract String grammarToState(String grammar, String oldstate);
abstract String stateToGrammar(String state);
```

4.3. Element Mapping

As *element mapping* we define the mapping between the element representation within the textual representation of the diagram (the grammar) and the representation within the element's property text. This distinction is necessary because in most cases the element text will be too complicated to display it within the diagram's text representation.

To ensure that both texts are in sync, the mappings have to be applied every time one side is modified. The following example will demonstrate the behavior:

Listing 4.2: Element mapping - diagram text

```
!!mycustomelement
```

This line within the grammar would map to an element named *mycustomelement*. The "!!" symbols are needed to distinguish the element type from other elements. Other than that, no additional element information is visible in the textual representation of the diagram.

Listing 4.3: Element mapping - element text

```
mycustomelement
additional information1
additional information2
bg=blue
fg=red
```

In contrary to the representation in the grammar the element does not need the "!!" symbols in the element text. Since no other elements need to be displayed in this view, the entire element data can be shown. In this case two additional lines of information are given, the background color is set to blue, and the foreground to red.

Listing 4.4: Element mapping - mapping code diagram to element

```
public String grammarToState(String grammar, String oldstate) {
  removeFirstLine(oldstate);
  removeFirstTwoCharacters(grammar);
  String state = grammar + LINEBREAK + oldstate;
  return state;
}
```

To map changes in the grammar text to the element text, two pieces of information are required. First, the new text used in the grammar is required. Secondly, the current element state, which is the state before the change occurred, is required. Within this example it is assumed that the element's representation in the grammar is mapped to the first line of the element state. The possibility that comments might be placed at the beginning is ignored.

The mapping code removes the old state's first line, including the two exclamation points, from the element's representation in the grammar. Then the new first line and the rest of the old state are concatenated and returned.

Listing 4.5: Element mapping - mapping code element to diagram

```
public String stateToGrammar(String state) {
  String firstline = getFirstLine(state);
  return "!!" + firstline;
}
```

The mapping from the element's state to the element's representation in the grammar is even simpler. Because the whole element information is visible within the element's state only one information—the new state—is required. The first line of this state is then concatenated with the two exclamation points.

Although this example is both simple and similar to a realistic implementation, these mappings have to be programmed by the user and might be overwhelming.

4.4. Template Alteration

The template alteration is very simple. The change just introduces the functions specified in section 4.2 and provides markers where the code should be inserted. The following example template can be used as long as no additional functions are required for other diagram types.

Listing 4.6: Custom elements (textual grammars) - example template

```java
import java.awt.*;
import java.util.*;

import com.umlet.constants.Constants;

public class TextualCustomElementImpl extends com.umlet.custom.
    TextualCustomElement {

  public void paint() {
    Vector<String> textlines = Constants.decomposeStrings(this.getState());
    /****PAINT START****/
    /****PAINT END****/
  }

  public String getDiagramType() {
    return
    /****TYPE START****/
    /****TYPE END****/
    ;
  }

  public boolean acceptIncomingConnections() {
    return
    /****INCOMING START****/
    true
    /****INCOMING END****/
    ;
  }

  public boolean acceptOutgoingConnections() {
    return
    /****OUTGOING START****/
    true
    /****OUTGOING END****/
    ;
  }

  public Point getDefaultCoordinate(Direction dir) {
    /****COORDINATE START****/
    if(dir == Direction.TOP)
      return new Point(this.width/2,0);
    else if(dir == Direction.RIGHT)
      return new Point(this.width,this.height/2);
    else if(dir == Direction.BOTTOM)
      return new Point(this.width/2,this.height);
    else
      return new Point(0,this.height/2);
    /****COORDINATE END****/
  }

  public String getElementExpression() {
    return
    /****EXPRESSION START****/
    /****EXPRESSION END****/
    ;
  }

  public String grammarToState(String grammar, String oldstate) {
    /****GRAMMAR2STATE START****/
    /****GRAMMAR2STATE END****/
  }

  public String stateToGrammar(String state) {
    /****STATE2GRAMMAR START****/
    /****STATE2GRAMMAR END****/
  }
}
```

4.5. UI Alteration

Because of the increased amount of methods that have to be implemented by the user, the original approach's simplistic user interface is no longer sufficient.

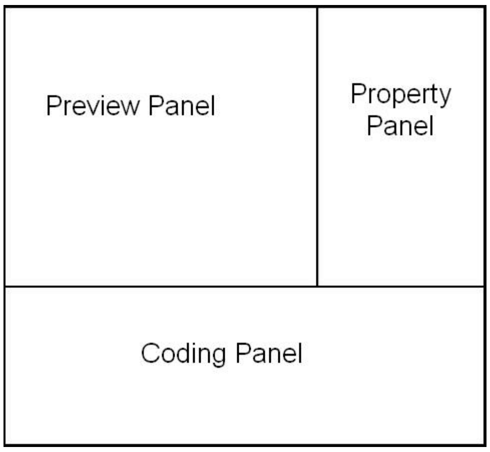


Figure 4.1.: Custom elements (textual grammars) - user interface

The main problem is the screen real estate required for displaying the preview panel because it does not only contain the element but an entire diagram containing the element. To create space, palette panels are hidden as soon as the custom element edit mode is entered.

The user interface's general layout is shown in figure 4.1. The code panel's size does not change even though the number of methods to implement increases. The reason for this is the coding process which is shown in section 4.5.2.

4.5.1. Preview Panel

The preview panel contains two figures. First it contains the usual representation of the element as a single element. This is useful as long as only the paint method is implemented because it is impossible to insert the element into a diagram at this point.

As soon as all methods are implemented a default diagram is displayed. The user can then insert the element into the diagram and test the element's behavior within the textual grammar. To support this, every grammar has to provide a default diagram that ideally contains all elements of this diagram type so that the new custom element can be tested together in the entire diagram's scope.

The preview and property panels together implement normal diagram behavior except that no additional elements can be added. This ensures that the element is tested under real circumstances while not confronting the user with elements that don't belong to the selected grammar.

4.5.2. Code Panel

The code panel shown in figure 4.2 consists of three different parts. Instead of only providing a simple coding panel it provides an element generation wizard that guides the user through the element generation process. Figure 4.2 shows the *general properties* step of the process as this is the only part that does not make use of a coding or help panel. The advantages of using a step-by-step process rather than displaying all methods at once are that there is more space for the preview panel and the user is not confronted with the full complexity at once.

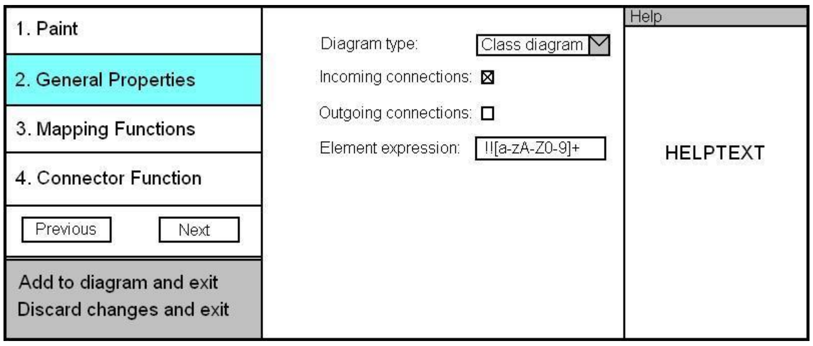


Diagram type: Class diagram
Incoming connections: ☒
Outgoing connections: ☐
Element expression: !![a-zA-Z0-9]+

Figure 4.2.: Custom elements (textual grammars) - user interface - code window

The left panel shows the current progress in element generation, shows which part is currently edited, and allows navigation between the different editing categories by the "next" and "previous" buttons as well as by clicking the different categories. This panel also contains buttons to finish or cancel the process.

The right panel provides a help text related to the current step in the element generation. This should help new users to become familiar with the various steps and provides background information about the utilization of the currently implemented method.

The center panel provides content for four process steps:

1. Enter code to paint the element
2. Specify general properties
3. Enter code to generate the mappings
4. Enter code to specify the connection points

The code entering steps only provide one or two (in case of step three) coding panels where the user enters the code to specify the required functionality. The single element preview is displayed as soon as step one is finished successfully. The next step is to specify the properties of the element that do not require the user to enter code (see section 4.2 for details).

After all steps have been completed the "Add element to diagram and exit" button becomes click-able and the user can complete the process.´

4.6. Conclusion

In this chapter the approach of user generated element (chapter 2) and text based diagrams (chapter 3) were merged together by enabling the user to integrate self generated elements into the text based diagram.

One drawback of this approach is the higher complexity of user generated elements due to the required mappings between the textual diagram's and element's representation. This drawback seems very negative at first glance but is not that serious because users may be introduced to custom elements by using normal custom element generation. If already familiar with the original approach the learning effort for including elements into a grammar is acceptable.

To sum up, this chapter introduced a powerful approach to integrate user generated elements into text based diagrams while keeping medium complexity through a good development structure and templates with a relatively easy entry point for the user.

5. UMLet Refactorisation and Feature Overview

As software evolves, its code quality decreases due to quick-and-dirty bug fixes and new features that do not exactly fit into the original architecture. Refactoring inverts this process of quality loss. It restructures the code so that the quality of the design is improved, ideally beyond its original level. If refactoring is applied constantly during the software evolution process, no loss of quality should occur.

Martin Fowler [16] defines refactoring as: "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." Refactoring may also be described as *cleaning up code* although it provides techniques to do that in a structured and efficient way.

The major goal is to get rid of so called *bad smells in code* [16]. These bad smells indicate the need for refactoring. The term *smells* is applicable because the decision when refactoring is both necessary and beneficial, is subjective. The following examples demonstrate the nature of such smells.

- Duplicated code
- Long method
- Large Class

A code fragment that exists more than once is an indicator for bad design and will obviously complicate maintenance. Long methods are methods that contain too much code to be understood easily. Large classes have too many variables and responsibilities, which usually violates the object oriented rule of high cohesion [9, 37].

To get rid of *bad smells* several refactoring patterns are provided. Some example patterns are:

- Extract Method
- Move Field
- Extract Class

The extract method pattern extracts lines of code into shorter methods with meaningful names. This makes the code much easier to understand and possibly avoids redundant code.

Moving fields between classes is more complicated. As software evolves, several new classes are added and some fields may become more appropriate in a new class than in the original class. Another good example for the need of moving a field is the Extract Class refactoring explained below.

A class grows during development. Usually this starts by adding some necessary methods and fields and ends up with a class having too many responsibilities to be understood and maintained easily. If such a class exists it is necessary to split the class into multiple classes to distribute the original class's responsibilities. The respective class members then have to be moved to their new classes.

Many of these patterns have been applied to UMLet. It was necessary because several persons have developed new features that did not exactly fit into the architecture. Other reasons were newly planned features that required higher cohesion and lower coupling of some architectural parts.

Although good refactorisation is a major goal, this chapter does not only focus on structural improvements but also describes new features. Section 5.1 gives a quick overview of the refactorisation goals and the most important requirements for the new features presented in section 5.2. Then section 5.3 gives an overview of UMLet's refactored architecture. This section's subparts focuses on those parts of the application where most refactorings have been applied. Section 5.4 lists open tasks related to not fully implemented features, and section 5.5 provides an outlook to the UMLet's future.

5.1. Goals and Approach

The main goals of this chapter are:

- (G8) Refactor UMLet's architecture to separate the components and increase the coupling/cohesion attributes of the code base; the result is evaluated qualitatively using OO-criteria and design patterns.
- (G9) Improve UMLet's integration into Eclipse.

In order to achieve better maintainability and extensibility (G8), architectural refactorisations are required. Two of the main problems in the UMLet 8.0 architecture are that the code to handle the drawing and user interaction with diagrams is split all over the applications code, instead of providing a unit of work. That was the reason why the application did not support concurrent diagram editing until version 9.0. Like the diagram logic, the user interface was not separated from the business logic of the program, which makes it difficult to change the user interface without also modifying the business logic.

Another problem is that there only exists one listener for all components (e.g. menu items or interactions with diagrams). In order to distinguish the various calling components (without tedious instanceof keywords), the global listener has to be split into several listeners, ideally one for each component.

The last problem is the complex custom element generation. The custom element generation technique used in version 8.0 requires the user to both define the entire class structure for the custom element and implement multiple methods. This might be overkill for an inexperienced user who has limited knowledge about the underlying architecture.

These problems lead to six refactoring steps which are defined as follows:

1. Apply various refactorings to the architecture
2. Separate user interface from business logic
3. Provide possibilities to alter / create user interfaces easily
4. Implement the diagram logic as one unit of work
5. Allow the user to edit multiple diagrams within one application window
6. Provide easier custom element generation

Step (1)—although separate step—also covers most of the other steps. Step (2) is a prerequisite for step (3) because it would be very difficult to provide a maintainable user interface if it inherently required adoptions in the business logic. The implementation of step (2) and (3) are discussed in section 5.3.6.

Similar to the steps mentioned above, step (4) is a prerequisite for step (5). Without a well designed diagram component, supporting multiple diagrams would become too complex. The diagram component as well as the multiple diagram editing functionality will be discussed in sections 5.3.1, 5.3.3 and 5.3.4.

Step (6) uses templates to reduce the complexity of the code exposed to the user. This approach is described in more detail in chapter 2. Some implementation details about it are provided in section 5.3.8.

In addition to these refactorings, several features (section 5.2) have been added/improved during this work (e.g. Copy/Paste). The most important of these was to improve UMLet's Eclipse integration (G9).

5.2. Feature Overview

Apart from implementing the goals, several features were implemented in UMLet version 9.x. While some target some of the goals mentioned above, others are relatively unrelated.

5.2.1. Edit Multiple Diagrams

The most important feature is to enable the user to be able to edit multiple diagrams within one application window. This feature is derived from goal (4) and introduces several benefits:

- Reduced memory load
- Increased usability
- Copying elements between diagrams

The smaller memory footprint is a minor improvement because today's high memory computers are usually able to handle many small applications like UMLet. The user is now able to close all diagrams at once and the navigation between different diagrams and other applications (e.g. an Internet Browser) is simpler as there are no unnecessary application windows. Also, copying of elements between diagrams is now possible provided that the diagrams are opened in the same application window.

5.2.2. One Application

Since it is now possible to edit multiple diagrams within a single application window, the next step is to ensure that additional diagrams are opened in existing application windows to prevent multiple application windows. This feature is discussed in detail in section 5.3.1.

5.2.3. Easy Palette Alteration

Another usability issue was that users were not aware of the fact that a palette can be edited like a diagram. To improve access to this functionality, a new menu-entry "edit current palette" has been added. This command opens the currently selected palette as new diagram so the user does not have to know where the palettes are stored in order to edit them.

When a currently used palette is saved, UMLet will automatically reload it in the palette-panel.

5.2.4. Copy/Paste

In order to allow element exchange between different diagrams, copy/paste functionality has been implemented. The copy command copies selected elements to the internal clipboard. The paste command inserts the contents of the internal clipboard into the current diagram. If the elements are inserted into the diagrams they were copied from, they are inserted at a position one inch to the right and bottom of the original elements. This ensures that copying elements within a diagram just behaves like duplicating elements. If the elements are inserted into another diagram they are inserted at the top left corner of the current view.

UMLet's copy operation also copies an image of the entire diagram into the global system clipboard for use by other applications.

5.2.5. Simple Custom Element Generation

Another goal (5) was to provide easier access to UMLet's custom element generation. In version 8.x the user had to implement a full Java class with several methods just to create a new custom element. Another issue was the lack of immediate feedback and unsatisfactory error management as the compiler error message was displayed to the user via a message box instead of highlighting erroneous code.

All of the aforementioned issues have been addressed in version 9. The amount of required user code for new elements has been reduced to a single method and compile feedback is now provided instantly with the code.

The new approach is described in detail in chapter 2 and some implementation details are provided in section 5.3.8.

5.2.6. Eclipse + Standalone UI

This feature targets goals (1) and (2). It was already in place in version 8.x but the user interface logic was not encapsulated in a single unit of work. Changing the behavior of any of two UMLet interfaces was therefore relatively complex. In conformance with the previously defined goals, the user interface logic is now implemented within a single unit of work and is therefore easier to maintain. New interfaces may be added without adapting the business logic and vice versa. The user interface component's implementation is described in more detail in section 5.3.6.

5.2.7. Platform Dependent Layout

The platform dependent layout comes with several advantages:

1. The end user is provided with a familiar environment;
2. The application's layout fits perfectly into the operating system's layout; and
3. Several platform dependent UI features are available (e.g. the platform specific file dialogs)

Furthermore this feature is rather easy to implement. The Java swing libraries used for UMLet the UI support platform dependent look & feels by setting a custom UI manager via the following line of code:

Listing 5.1: Adding platform dependent layout to swing

```
import javax.swing.UIManager;
...
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

5.2.8. Eclipse Integration

Although UMLet has been integrated into Eclipse in previous versions, the only Eclipse integration points were Eclipse initiated file saving and loading operations.

In order to improve the usability within Eclipse, several features have been added. First the menu bar within the diagram window has been removed and integrated into the Eclipse menu bar and then common functions (e.g. redo, undo, copy, paste) of the Eclipse edit menu were added to support the UMLet plugin.

5.2.9. Configurations

The reasons for adding configuration files are obvious.

1. Provide the user with the option to modify default values of global properties like font-size
2. Persist global properties so that the properties can be applied at subsequent application launches

While this might be a trivial feature to implement, it greatly improves the quality of service to the end user and avoids confusion at application start-up.

5.2.10. Search

The improvement of the search mechanism is very simple. Instead of selecting all elements that matched the search string, the first found element is selected and the user interface ensures that the current view is moved to that element.

This makes the search more meaningful since searches are more likely to be performed on large diagrams that do not fit on the screen.

5.2.11. Easy Access

Easy access to the application for first time users was not given until version 9. The application could only be started from a jar- or batch-file in a subdirectory of the application.

To provide easier access, an UMLet.exe (for windows), UMLet.sh (for linux), and UMLet.jar is provided in the root directory of the application. These files only launch the application file in the subdirectory but provide easier access to the application for first time users.

5.3. Architecture Description

In this section we will give a short overview of the different components and their interactions within version 9.x of UMLet. The architecture of UMLet generally consists of eight packages that are visualized in figure 5.1.

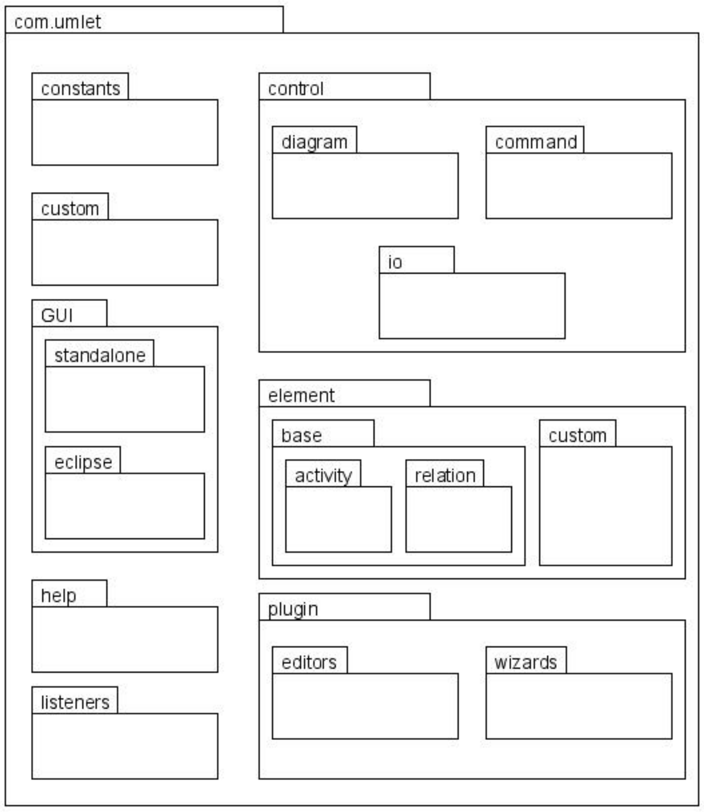


Figure 5.1.: Package diagram

The package *com.umlet.constants* contains several constants and shared static methods that are used in many other parts of the application. Some of these constants are loaded from and saved to a configuration file (see section 5.3.2). The most important methods take care of some graphical layout properties (e.g. font size) or the decomposition of strings (line splitting and comment removal).

The package: *com.umlet.custom* takes care of the custom element generation engine (see chapter 2 and section 5.3.8). It contains all methods for element generation, implements error handling, and provides annotations for the definition of supported methods and variables in the source code panel.

The *com.umlet.help* package contains several UI elements (e.g. the about dialog) and loads textual help contents from external html files on the file system.

The last package without sub packages is the *com.umlet.listeners* package that contains all of UMLet's event listeners used for diagram handling. The listeners used in individual graphical user interfaces (e.g. listeners for menu interaction) are located in the appropriate GUI package.

The package containing all supported UML elements is named *com.umlet.element* and split into two packages. The first one (*com.umlet.element.custom*) contains elements that were initially developed by users and then added to the distribution. The second one (*com.umlet.element.base*) contains all basic elements designed by the UMLet programming team, including relations between elements in the *com.umlet.element.base.relation* package and the all-in-one activity diagram in the *com.umlet.element.base.activity* package.

The most important package is the *com.umlet.control* package. It contains UMLet's business logic. While the basic logic and the startup routines (section 5.3.1) are contained in the com.umlet.control package itself, the logic behind the diagrams is implemented in the *com.umlet.control.diagram* package. The logic needed for file operations is located within the *com.umlet.control.io* package. The last subpackage (*com.umlet.control.command*) contains the commands (section 5.3.5) that may be executed by the user and also handles the do/undo logic.

When the application is started as an Eclipse plugin, the *com.umlet.plugin* package handles the application launch (instead of the com.umlet.control). The subpackages *com.umlet.plugin.editors* and *com.umlet.plugin.wizards* are responsible for the embedded UMLet editor and the Eclipse UMLet file creation wizard. The plugin mechanisms used are described in detail in section 5.3.7.

Depending on how the application is started, certain GUI packages will be used to build the user interface. General GUI components used for both modes are contained in the *com.umlet.GUI* package. Components that are only used in one of the two modes are contained either *com.umlet.GUI.standalone* or *com.umlet.GUI.eclipse* packages. Detailed information about the GUI separation is available in the sections 5.3.6, 5.3.6, and 5.3.6.

5.3.1. Application Startup

During application startup we differentiate between three different modes shown in figure 5.2.

In the first mode the application will be started normally. Additionally a method that listens for new diagrams to be opened will be started.

If the application is started with a "-filename=..." command line argument it first tries to find a running application in which the file could be opened in a new tab. It only starts the application

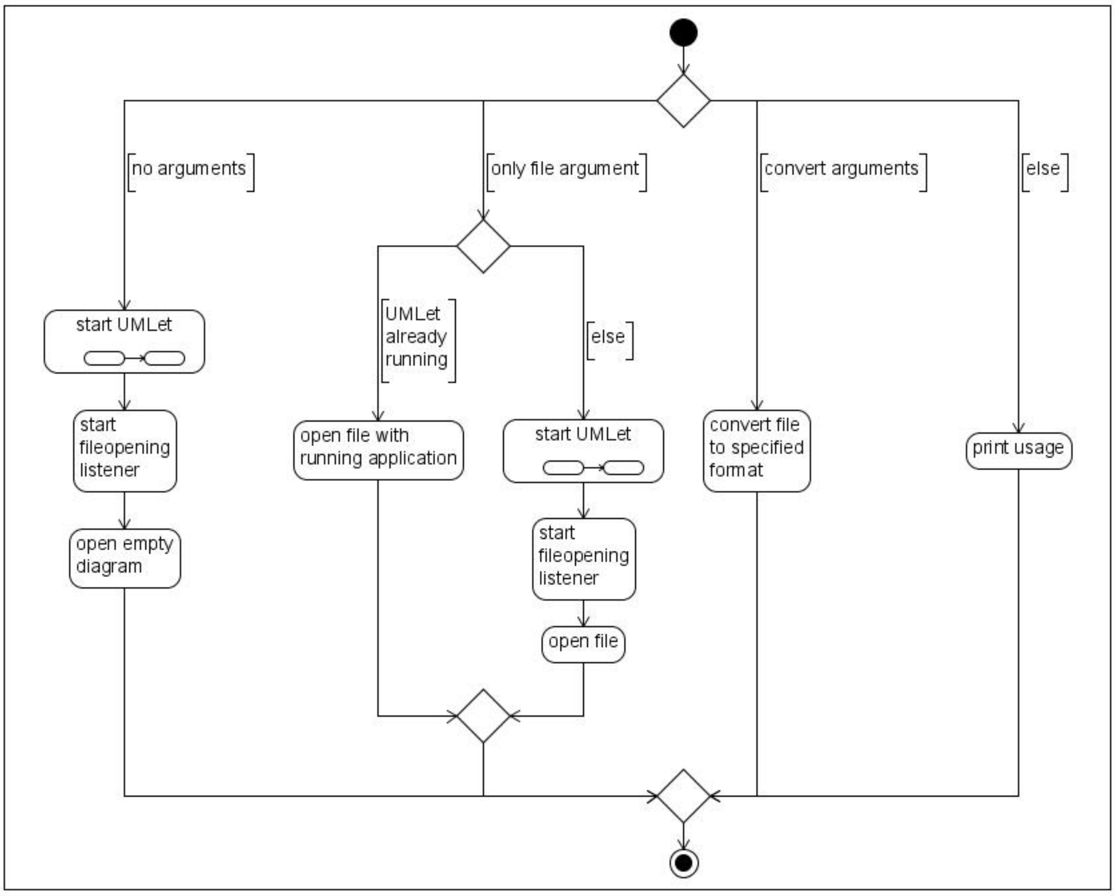


Figure 5.2.: Application startup

if no other UMLet instance is open.

If started in the third mode, UMLet will convert one file to a specified format and terminate.

The application start for Eclipse is simpler as Eclipse handles opening of diagrams and provides mechanisms that ensure that only one instance of the application is running within one instance of Eclipse. The Eclipse integration will be discussed in more detail in section 5.3.7.

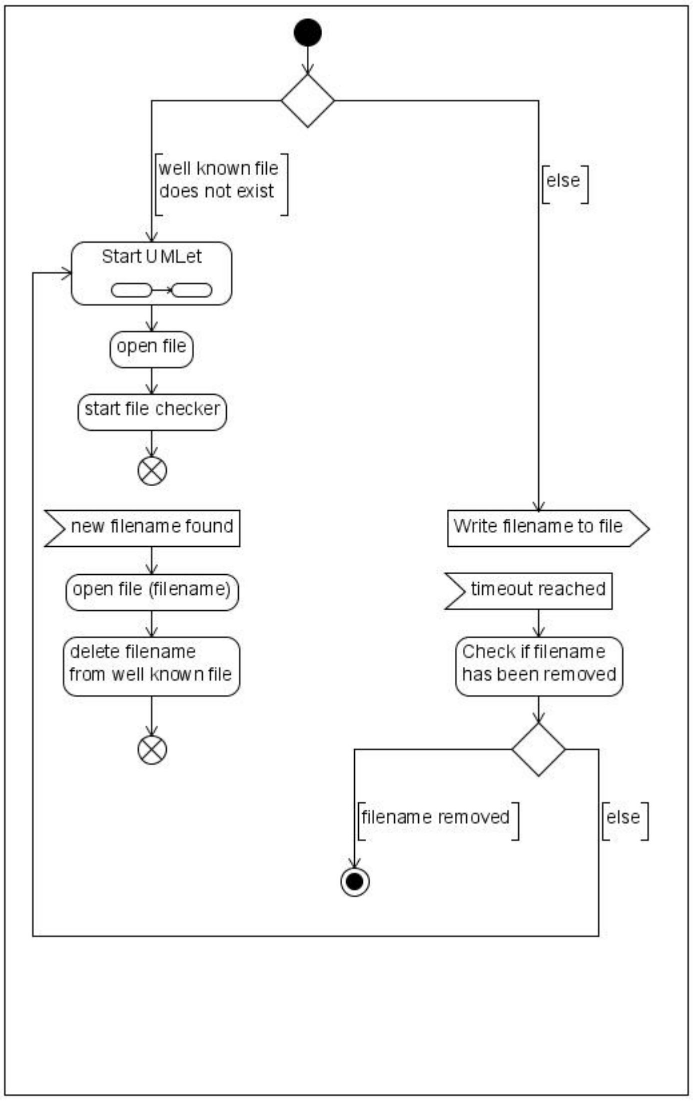


Figure 5.3.: Application startup - common file

Inter process communication

There are several implementation options to provide the *inter process communication* that is required for transmitting the file references:

1. Java RMI;
2. Sockets; and
3. Common file

Java RMI is the most powerful paradigm because it supports remote method invocation. However, for the purpose of transmitting a single file, it would add unnecessary complexity.

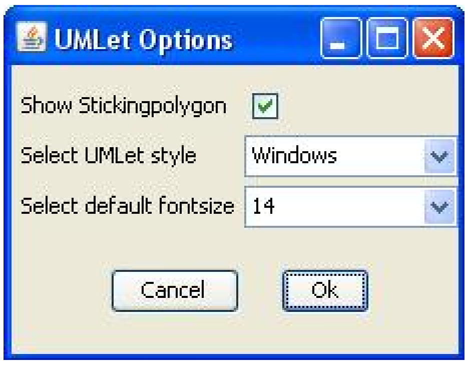
Sockets are an easy solution for transmitting a single line of text to a running application but they require an open network port to be able to listen to new incoming filenames. Another drawback of sockets is that most firewalls treat open serversockets as attempt to access the internet, which would be a false alert in this case.

Therefore, the best solution for checking if UMLet is already running is to create an empty common file (e.g. umlet.tmp) during startup. If the file already exists, the filename of the file to be opened is stored within that file. The already running UMLet instance checks the file periodically and opens files that are listed in umlet.tmp. After a file has been opened, the filename is removed from umlet.tmp and the application continues checking for new filenames. Finally, the application deletes the common file before it terminates. This process is visualized in figure 5.3.

One problem of this approach is that the common file may not be deleted if the application that created the file does not terminate properly. To overcome this issue, the application sending the filename checks if the filename has been removed by a running UMLet instance. If not it assumes that there is no running UMLet instance and performs a normal UMLet startup itself.

5.3.2. Configurations

The configuration panel currently features the three options shown in figure 5.4.



Show Stickingpolygon ✓

Select UMLet style: Windows

Select default fontsize: 14

Figure 5.4.: Option panel

In addition to user settings, UMLet's layout is stored in the application configuration file. A sample configuration file is shown below.

Listing 5.2: Configuration file - example content

```
default_fonsize = 14
show_stickingpolygon = true
ui_manager = com.sun.java.swing.plaf.windows.WindowsLookAndFeel
main_split_position = 738
right_split_position = 561
start_maximized = false
umlet_size = 1249,1036
umlet_location = 18,28
```

These values are stored in the configuration file as soon as the application terminates and are loaded during UMLet's startup procedure ("start UMLet" activity) as shown in figure 5.2.

5.3.3. Diagrams

One of the major differences between UMLet's version 8.x and 9.x is that diagrams were not implemented as a unit-of-work in version 8.x. The multiple diagram editing feature in version 9.x required to implement it as a separate component. The general structure of a diagram is shown in figure 5.5.

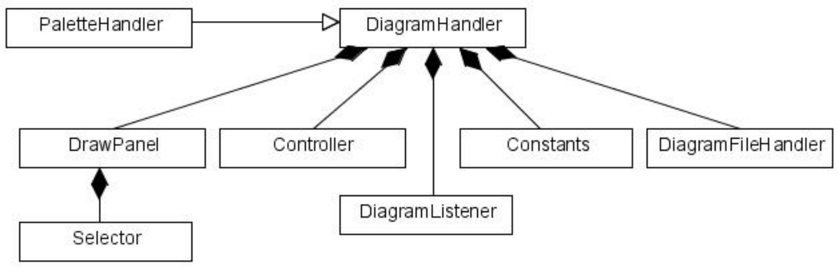


Figure 5.5.: Diagram - unit-of-work

The central class is the *DiagramHandler*. It represents the diagram itself and manages the interactions between other, more specific classes. The *DrawPanel* is the panel that contains the painting area on which the elements are drawn. The *Selector* keeps track of selected elements within the diagram. The *Controller* maintains a list of executed commands to enable undo and redo functionality for each diagram. The *Constants* class stores configurations (e.g. font-size) which are used by several components. The *DiagramFileHandler* models the diagram's file representation and provides methods to save the diagram into various formats.

The *DiagramListener* listens to mouse actions that occur on the drawing panel and invokes the respective commands where applicable.

The DiagramHandler can be extended to provide additional, change, or restrict diagram functionality for other types of drawing panels (e.g. palettes).

5.3.4. Listeners

The listeners shown in figure 5.6 are responsible for handling mouse events that occur on the diagram and it's elements(i.e. *Entities*).

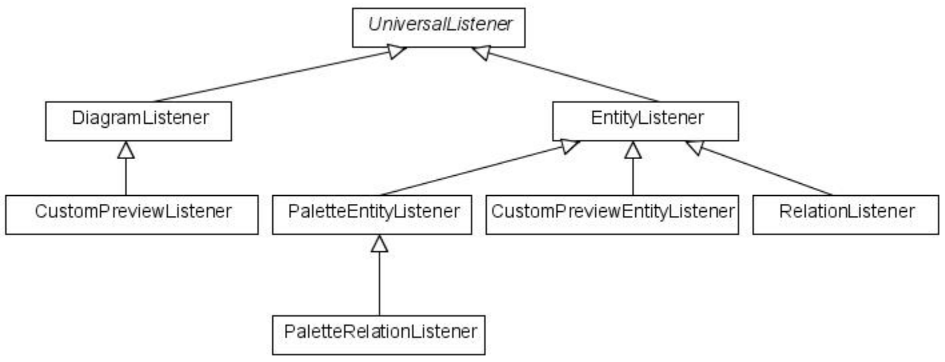


Figure 5.6.: Diagram - listeners

Every listener adds to or modifies the *UniversalListener*'s behavior. The *DiagramListener* is responsible for interactions with the diagram itself (e.g. clicks on the diagram area) while the *EntityListener* takes care of interactions with the elements placed on a diagrams (e.g. moving an element). The information shown below documents the added or modified functionality for various listeners in respect to their respective superclasses.

UniveralListener

The UniversalListener implements functionality that is used by both the diagram itself and its elements.

Every time an object is clicked, the listener designates the object's diagram as the currently edited diagram in the GUI. This ensures that the command functionality as well as the property panel operate on the currently active DiagramHandler.

If the mouse is dragged while a selector frame is active, the size of the selector frame is adjusted accordingly. Once the mouse button is released, all elements within the selector frame are marked as selected.

DiagramListener

The DiagramListener extends the Universallistener and adds the following behavior.

When the mouse is dragged, the entire diagram is moved. If a mouse button is pressed without an active CTRL key, all elements are deselected. With an active CTRL key, a selector frame is set up at the current mouse position. When the mouse is moved and not positioned on an element the cursor changes to the default cursor.

CustomPreviewListener

The CustomPreviewListener is identical to the DiagramListener but has been implemented to allow for future custom functionality for either the normal diagram panel or the custom element preview panel.

EntityListener

The EntityListener extends the DiagramListener by adding element operations like select, move, resize and duplicate. When the mouse is clicked, the element is selected and a double-click causes the element to be duplicated. Additionally, whenever the mouse moves inside an entity's bounds, the mouse cursor is changed to either the hand-cursor or the appropriate resizing cursor. If the mouse is dragged, the element is either moved or resized depending on the position where the dragging started. Additionally the EntityListener checks whether relations stick to an element and thus should be adjusted accordingly.

Another extension to the DiagramListener is the context-menu that contains options to change the element's color or to delete the element. The context menu is shown when the mouse is right-clicked.

RelationListener

The RelationListener adapts the EntityListener's behavior to accommodate the movement concepts of relations. Every vertex of a relation can be moved separately. In addition the whole relation can be moved if dragged at a specific position which is marked with a symbol. If a line is dragged without moving a specific vertex or the whole line a new vertex is created at the moved position.

Another adaption is the customization of the displayed mouse-cursor depending on which area of the relation the mouse is moved.

PaletteEntityListener

The PaletteEntityListener only changes one aspect of the EntityListener. If an element is double-clicked, it will not be duplicated on the same diagram but is instead added to the currently active diagram—or nowhere if no diagram is available.

PaletteRelationListener

The PaletteRelationListener extends the PaletteEntityListener with the same functionality that the RelationListener adds to the EntityListener.

CustomPreviewEntityListener

The CustomPreviewEntityListener suppresses the EntityListener's double-click functionality to prevent the user from duplicating custom elements in the preview panel.

5.3.5. Commands

The command pattern is applied to every action that is invoked on either a diagram or an element in order to support undo/redo functionality. The class diagram in figure 5.7 outlines the general structure of the commands in UMLet.

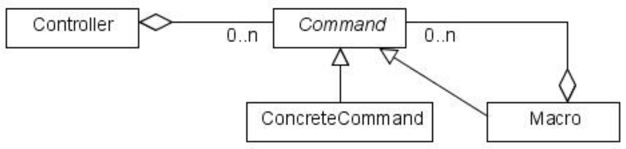


Figure 5.7.: Commands

The *Controller* executes every command that is added to its list. Commands that should not be undone/redone must not be added to the controller and have to be executed independently. Examples are composite commands or implicit commands like those of self resizing elements. Thus the controller should only be used for commands triggered by user interaction which implies the use of listeners (section 5.3.4).

Every time the controller executes a command, it attempts to merge adjacent commands. If suitable commands (e.g. two move commands) are found, they are merged into one new command that covers both. Thus only one click is necessary to undo both actions. Another special command is the *Macro* which may be used to execute several commands, including commands that cannot be merged, within a single action.

The following subsections give quick overview of the available commands and their respective redo/undo behavior as well as details on which commands can be merged.

AddEntity

The *AddEntity* command adds a new element at a specified position to the diagram.. This feature may also be used to add an element to an entity either by adding the element from another diagram or by duplicating it in the same diagram. In either case, the element instance has to be cloned to be able to add it again as elements have to be distinguishable. Newly added elements are automatically selected.

The *AddEntity*'s undo action is to remove the element from the diagram by calling the *RemoveElement* command.

AddGroup

The *AddGroup* command combines the currently selected elements in one group that behaves like a single element. The *undo* action ungroups the elements.

ChangeBGColor

This command sets an element's background color to a specific color and reverts the change in case of an *undo* action.

ChangeFGColor

This command behaves similar to the *ChangeBGColor* command but instead changes an element's foreground color.

ChangeState

The *ChangeState* command alters an element's textual state. Usually this command is invoked when the user modifies an element's state using the property panel. Two or more *ChangeState* commands can be merged as long as they operate on the same entity.

The *undo* action resets the element's state to its original state, i.e. the state before the element was altered. If two or more *ChangeState* commands were merged, the original state is the state prior to the first *ChangeState* command.

Copy

The *Copy* command copies the selected entities into the internal clipboard and stores an image (bitmap) of the currently edited diagram in the system wide clipboard. The *undo* action has no effect.

Cut

The *Cut* command works just like the *Copy* command except that it also removes the selected elements from the current diagram. The *undo* mechanism restores the removed elements but does not empty the clipboards.

HelpPanelChanged

The *HelpPanelChanged* command is called as soon as the help text within the property panel is altered. This text is stored for the current diagram. If a font-size has been specified by the user, it is updated. This command can be merged with other *HelpPanelChanged* commands. As soon as an *undo* is executed, the text and font-sizes are reverted.

InsertLinePoint

The *InsertLinePoint* command inserts a vertex into a relation. The *undo* action removes the vertex.

Macro

A *Macro* is not a single command but rather a set of commands. Any command can be added to a macro, even other macros. The commands are executed in the order they have been added. The *undo* action simply calls the undo methods for every command stored in the macro in the opposite order they were executed. *Macro*commands may be merged with other macro commands as long as they contain the same number of commands that can be merged pairwise.

Move

The *Move* command moves an element or a relation from one position to another. The *undo* action moves the element back to its original position. *Move* commands may be merged as long as they operate on the same element. The move command does not move adjacent relations as this is the responsibility of the calling method.

MoveLinePoint

The *MoveLinePoint* command implements the *Move* command for vertices or end points of relations.

Paste

The *Paste* command adds elements that were copied to the clipboard to the diagram. If no entities are available nothing happens. The *undo* action removes previously inserted elements.

RemoveElement

The *RemoveElement* command removes a group of elements or relations from a drawpanel. The *undo* action restores the elements.

RemoveLinePoint

The *RemoveLinePoint* command removes a relation's vertex and should be utilized when a vertex becomes obsolete (e.g. by moving it into the direct line between its adjacent neighbors). The *undo* action restores the vertex at its original position.

Resize

The *Resize* command resizes an element in one or two dimensions but does not resize ore move adjacent relations as this is the responsibility of the calling method. The *undo* action revertsthe resize operation.

Search

When a diagram is searched, all states of all elements are taken into account, including hidden information. Matching elements or groups containing matching elements are selected. If none of the selected entities are within the current view of the diagram, the view is moved to ensure visibility of at least one of the found elements.

Ungroup

The *Ungroup* command splits up a *Group* and is equivalent to the *Group* command's undo action. The *undo* action for the *Ungroup* command restores the original group.

5.3.6. GUI Separation of Concerns

Separation of user interface logic and business logic is a common design pattern to allow for more flexibility in either layer, and especially for providing different graphical user interfaces for the same underlying logic.

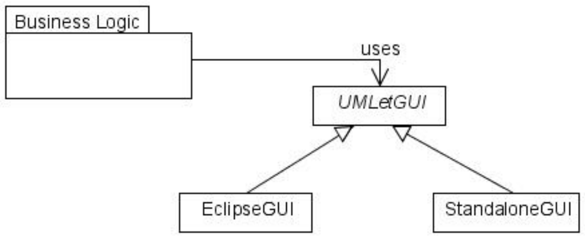


Figure 5.8.: UI separation

Therefore the user interface is defined in one basic abstract class (*UmletGUI*). The business logic only calls user interface methods in this class which makes it possible to create a new user interface by simply extending the *UmletGUI* class. To avoid a wrapping layer on the business logic side, the user interface may use common business classes (e.g. *DiagramHandler*, or *Controller*) directly.

To ensure an unified layout of the core components, the following modules are defined within the abstract user interface:

- The property text pane
- The context menu (if an element is right clicked)
- The drawing panel—for diagrams or palettes
- The whole custom element generation panel

- The tab components

The property text pane contains the label for the pane as well as the appropriate text field with the appropriate scroll panes and functions to mark special commands like the "bg=..." command.

The context menu contains several options to interact with the currently selected element. Depending on the element's type the user can : group, ungroup, delete, change back- or foreground-colors, or edit the element as custom element.

The drawing panel provides UMLet's basic drawing functionality and automatically attaches the appropriate listeners.

The custom element generation panel contains a property panel, the code panel, and the preview panel.

The tab components are closing buttons that are designed to be added to a swing component named *JTabbedPane*.

Figure 5.9 shows the different components as they appear in UMLet's standalone GUI. As outlined in the screenshot, the components cover most of UMLet's standard functionality. Therefore, the only purposes of implementing new user interface extensions are the arrangement of the components, the ability to switch between multiple diagrams, and the addition of menu elements.

Stand-alone GUI

As shown in figure 5.9 the standalone GUI consists of menu items, a panel containing a tab pane for an arbitrary amount of diagrams, a panel for the currently selected palette, and a custom code panel which is only visible if an element is edited or created.

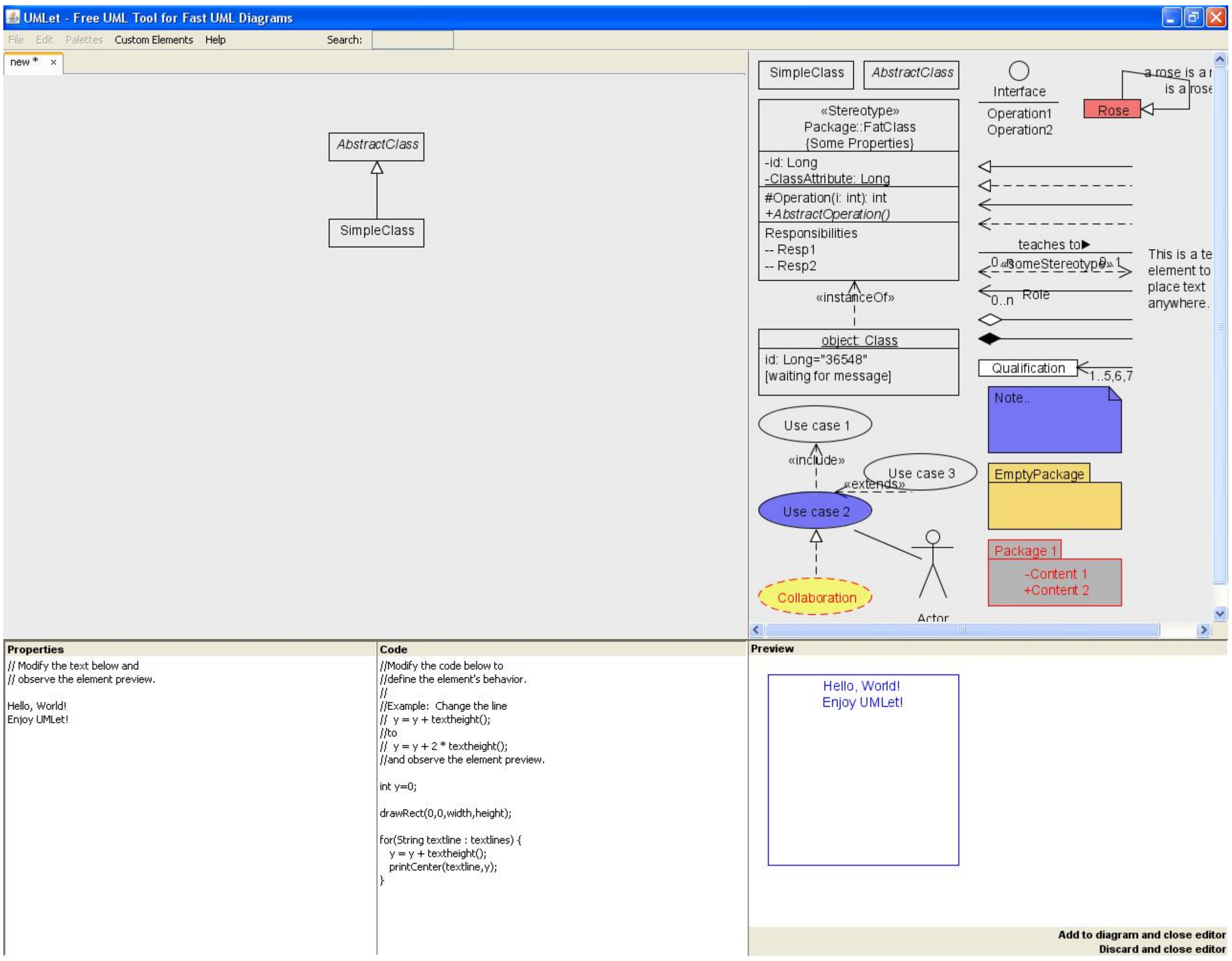
A diagram is opened via menu item. The palette can be selected using the palette menu entry. The UI class also implements a feature that disables menu items if they are not accessible at the moment (e.g. delete cannot be called if no items are selected).

The UI component itself represents the top panel which is directly added to the application frame which is why the UI class represents the user interface both logically and graphically.

Eclipse GUI

The Eclipse GUI slightly differs from the standalone UI.

The menu provided in the standalone version is completely integrated into Eclipse. The actions available within the "Edit" menu are integrated into Eclipse's "Edit" menu and the other


UMLet - Free UML Tool for Fast UML Diagrams
File Edit Palettes Custom Elements Help
Search:
new *
AbstractClass
SimpleClass
Preview
Hello, World!
Enjoy UMLet!
Add to diagram and close editor
Discard and close editor

Properties

```
// Modify the text below and
// observe the element preview.

Hello, World!
Enjoy UMLet!
```

Code

```
//Modify the code below to
//define the element's behavior.
//
//Example:  Change the line
//  y = y + textheight();
//to
//  y = y + 2 * textheight();
//and observe the element preview.

int y=0;

drawRect(0,0,width,height);

for(String textline : textlines) {
   y = y + textheight();
   printCenter(textline,y);
}
```

Figure 5.9.: UI - standalone

menus are grouped into a "UMLet" menu, which is only visible when an UMLet diagram is currently active.

Because of the lack direct accessible menus, the palette is accessible via tab panes rather than via menu.

Another difference is that the UI class itself has no graphical representation in Eclipse because diagrams are opened directly by Eclipse which prevents the UI class from representing a wrapping panel.

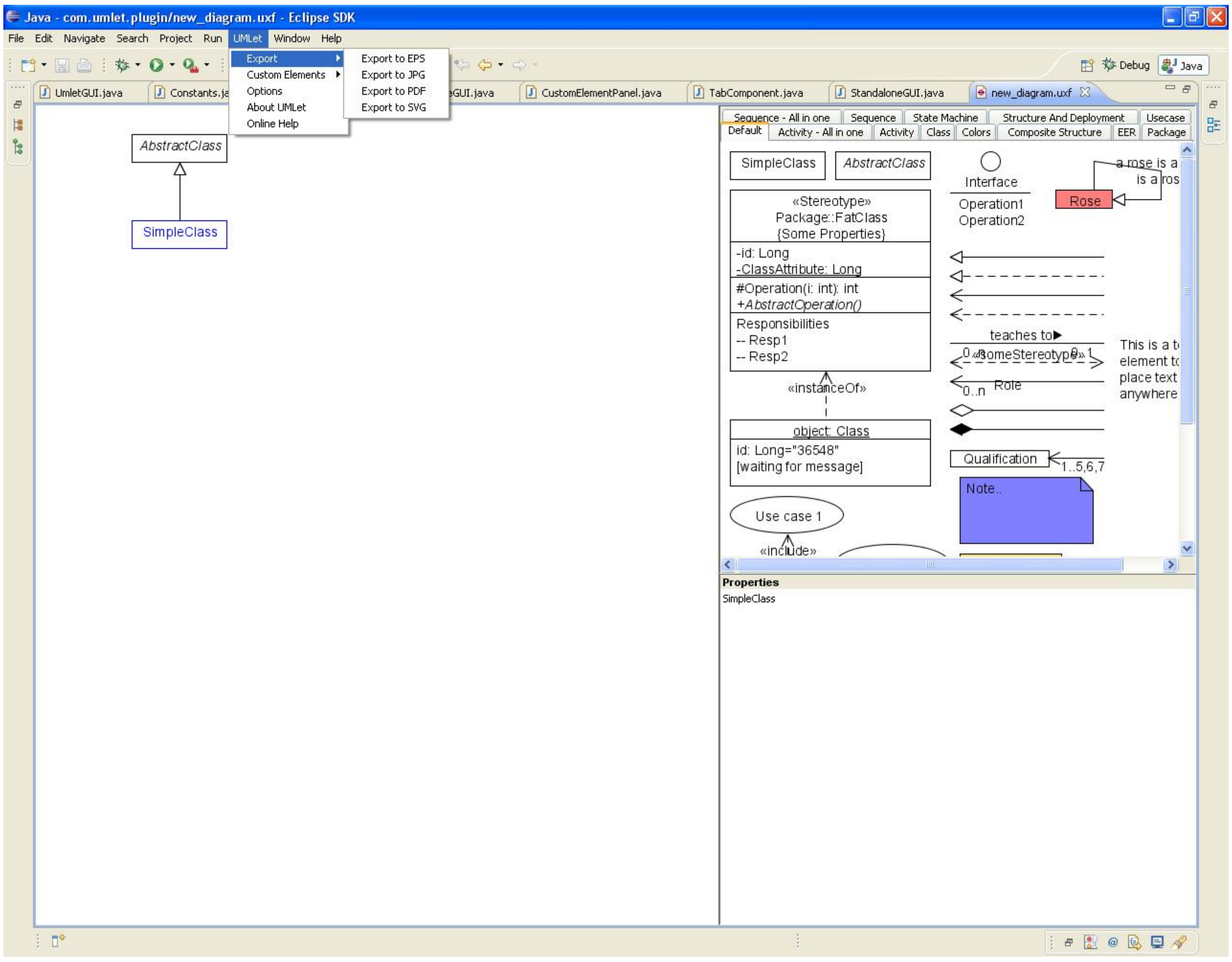


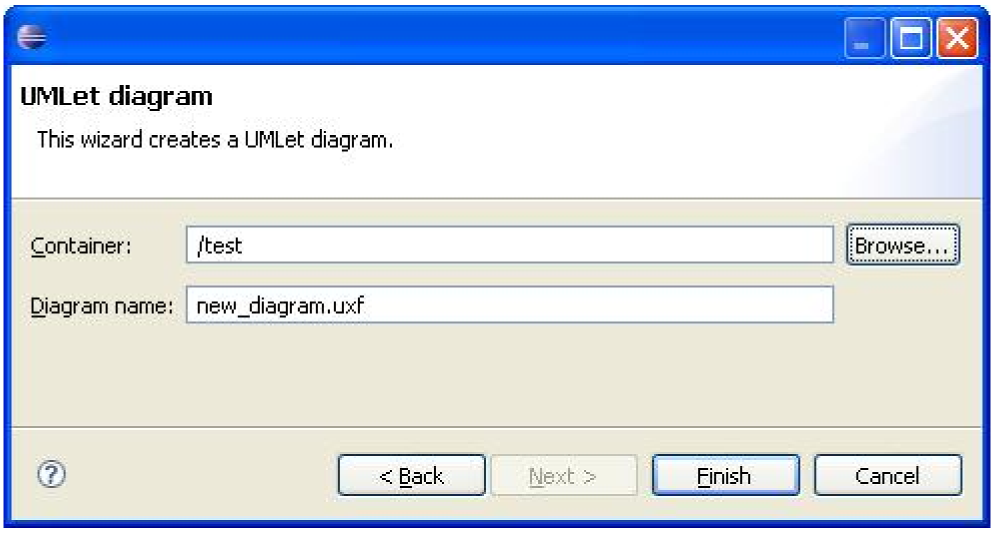
Figure 5.10.: UI - Eclipse

5.3.7. Eclipse Integration

The UMLet Eclipse integration generally consists of four parts:

- The plugin activation
- The diagram creation wizard
- The editor
- The menu contribution

The plugin activation is the simplest part. It simply starts the umlet application and initializes the Eclipse GUI logic which has (in contrary to the standalone GUI) no graphical representation.


UMLet diagram

This wizard creates a UMLet diagram.

Container: /test

Diagram name: new_diagram.uxf

Figure 5.11.: Eclipse wizard

The next part is the diagram creation wizard. It displays a single page dialog where the user can specify the folder and the name of the new diagram. A sample screenshot is provided in figure 5.11.

The editor is the main part of the UMLet application in Eclipse. It is instantiated as soon as a diagram is opened through the Eclipse environment. A new graphical component is allocated and the file is opened with the UMLet diagram application. This part is also invoked as soon as the diagram is saved or closed. An aspect that increases the complexity of the UMLet integration into Eclipse is that UMLet uses AWT components (Java swing) while Eclipse is built on the SWT component technology. Therefore, a SWT-AWT composite frame has to be used to accommodate the UI. The problem is that both SWT and AWT use their own execution stack for refreshing graphical components. Thus calling functions have to be used for calling swing function from SWT components and vice versa. The following code listing shows two examples.

Listing 5.3: Eclipse: SWT - AWT calls

```
javax.swing.SwingUtilities.invokeLater(new AWTAction());
org.eclipse.swt.widgets.Display.getDefault().asyncExec(new SWTAction());
```

The menu contributor is responsible for creating and maintaining the UMLet menu and its submenus. It also defines the actions that are executed when common Eclipse menu items like undo/redo or delete are called. Because of the SWT-AWT problem mentioned above, Eclipse cannot distinguish which component calls the action which is why every actionwill be executed on the current selected component even if a shortcut is utilized when editing the element within a text field. As a consequence, the element will be deleted in addition to removing the according character if the user presses the delete key. To avoid this problem, the Eclipse GUI disables the according menu items as soon as a text editing field is focused.

5.3.8. Custom Element Generation

The custom element generation interface of UMLet is shown in figure 5.9 and is discussed together with the custom element generation process in chapter 2.

5.4. Open Issues

This section lists several open issues that have not been addressed in the latest UMLet version but might be implemented in future UMLet versions.

5.4.1. Publishing Process

Currently there is no defined publishing process for UMLet updates. Such a process would be important to provide stable and consistent UMLet releases. Problems that occur without such a process in place include:

- Inconsistent version numbers between the Eclipse and standalone versions
- Continual time consumption for each programmer to get things right
- Bugs on other platforms

A short proposal for a publishing process addressing these issues is shown in figure 5.12.

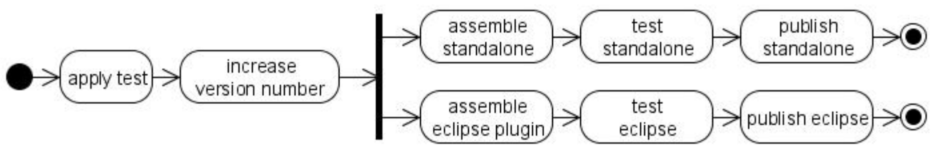


Figure 5.12.: Publishing process

First a testing scenario has to be set up. This is described in detail within section 5.4.3.

After the tests ran successfully, the version numbers should be increased and reflect the scope of changes. Currently the version numbers have to be changed in multiple locations:

- In the plugin information
- In the UMLet folder name
- In the "com.umlet.plugin/html/aboutumlet.html" file

Then the application has to be packaged as both a standalone application and a Eclipse plugin.

To create the standalone version the application has to be packaged as an executable jar file which is in turn copied into the "com.umlet.plugin" folder. Additionally, new libraries have to be copied into the "com.umlet.plugin/lib" folder. One important action is to zip the file on a non-windows system because if it is zipped on a windows system the UMLet.sh file gets corrupted.

To create the Eclipse version, the application has to be exported as an Eclipse plugin and the library and plugin settings have to be up to date.

After the applications have been assembled, the basic tests should be applied again. After a successful run, the packages may be published on the respective web sites.

5.4.2. Security

The custom element generation feature may be used by users to create malicious diagram elements that may harm remote systems. To overcome these issues, several security aspects have already been addressed. In general two security vulnerabilities are still pending:

- Not all methods defined as final
- No security manager within Eclipse

The first problem is that the custom class is able to override methods that are not secured by the Java security manager. This is possible because the methods are not defined as "final". The problem with defining the methods as final is that Java implementations slightly differ on the various platforms. Some methods are only available on some platforms. Because of that only a subset of methods can be secured without publishing the application separately for each platform. One solution for this problem would be:

- Check the implemented methods of the class by reflection and throw a security exception if a method other than the paint method is implemented.
- Change the design so that the custom element class only contains a paint method and does not inherit from another class. This class has to be called from a wrapping component.

The second problem is that the Java security manager can not be applied to Eclipse. As a consequence almost every operation can be performed within Eclipse, including the deletion of the entire file system. As of now, there is no solution for this problem.

A simple solution for both problems would be to inform the user that a diagram contains custom elements and asks if the diagram should be opened. This ensures that the user is responsible if he opens a diagram that contains malicious elements. Although this is no real security solution it is commonly applied in many tools that allow user written code to be executed.

5.4.3. Testing

Currently no testing scenario is in place. This requires the programmer to test the changes which usually results in overlooked bugs. A basic test plan should be generated to provide for basic regression testing before a release. Even the most basic test list could vastly improve UMLet's quality, although automated test cases (e.g. JUnit) are preferable.

Obviously, these tests have to be maintained as part of the UMLet project to ensure that both new and old functionality is covered.

5.4.4. Palette Editing in Eclipse

Currently the easy palette editing feature is only available in the standalone version of UMLet. It has not been implemented within the Eclipse version because Eclipse did not support the opening of external files (i.e. files that are not part of a project) until version 3.3. The following code is an example of how the feature could be implemented.

Listing 5.4: Eclipse issues - opening external files

```
File fileToOpen = new File("externalfile.xml");
if (fileToOpen.exists() && fileToOpen.isFile()) {
    IFileStore fileStore = EFS.getLocalFileSystem().getStore(fileToOpen.
        toURI());
    IWorkbenchPage page = PlatformUI.getWorkbench().
        getActiveWorkbenchWindow().getActivePage();

    try {
        IDE.openEditor...
    } catch ( PartInitException e ) {

    }
}
```

In addition to this code a new menu item which adds the code has to be created. The implementation should be added as soon as the support for Eclipse 3.2 or lower is suspended.

5.5. Outlook

Aside from the open issues mentioned in section 5.4 several new features will be implemented in future UMLet versions. To give a quick overview of these features, the most important ones are mentioned here:

- Diagram zooming
- More export formats
- More export options (landscape / portrait)
- Support toggling of palette panel
- Support inserting images into diagrams
- Support of storable workspaces (position and sizes of panels)
- More/extended "All-in-one" diagrams
- Option to display the Grid
- Remove Grid and instead provide a snap in functionality for relations
- Automatically resized elements
- Online repository for user created elements, diagrams, palettes.
- Fit other diagram types than UML
- UMLet as Webapplication

The first six features are relatively easy to implement and can be added to UMLet within the next major release if all stakeholders agree to add them.

The *All-in-one* diagram feature is described in detail in chapter 3. This would require a huge amount of work and major changes to the diagram logic described in section 5.3.3.

The options of either grid display or providing snap in functionality are exclusive to each other. It has not yet been decided which approach will be implemented.

Currently it is already possible to generate automatic resized elements with the custom element generation engine. It is also possible to generate elements that are resized according to its contents by default but can be resized manually too. Although this is already possible, the existing elements have not been updated yet, which may be a non trivial task because of the high complexity of some of these elements.

The online repository would help to create a better UMLet community and would make it possible to use UMLet for diagrams other than UML as new elements could be created and published by users so that the whole community would be able to use them.

An UMLet Webapplication would provide easy access for users searching for an UML tool that fits to their needs, which may increase UMLet's popularity.

After all UMLet is on the right way and improves with every version, but there is still a great amount of work to be done—and likely will always be.

6. Discussion

The goals of the different chapters were:

- (G1) Enable users to generate customized elements. (page 10)
- (G2) Avoid a media break when creating custom elements within UMLet–the custom element interface should be provided as part of the main application. (page 10)
- (G3) Provide full access to modify the elements' behavior, instead of a stripped-down macro language. (page 11)
- (G4) Provide a way to syntactically define activity diagrams, while avoiding interactive dialogs. (page 28)
- (G5) Provide a theoretical approach to spread the diagram by text generation approach to other UML diagram types. (page 28)
- (G6) Quantitatively evaluate the efficiency of this diagram creation process with a performance evaluation based on Tamir's GUI metrics. (page 29)
- (G7) Provide a theoretical approach to design user customized elements for textual grammars. (page 53)
- (G8) Refactor UMLet's architecture to separate the components and increase the coupling/cohesion attributes of the code base; the result is evaluated qualitatively using OO-criteria and design patterns. (page 65)
- (G9) Improve UMLet's integration into Eclipse. (page 65)

Goal (G1) and (G2) have been reached completely. Users are now able to generate customized elements in an interface that only adds one additional text field and one preview area to the usual user interface. These areas have been placed below the drawing area and thus are part of the main application.

Goal (G3) has been reached partially. The user has full access to the elements drawing behavior, but due to security reasons the user is not able to use file operations (e.g. loading an image into an area of the element) in the elements' custom code.

Goal (G4), (G5), (G6) have been reached entirely. Users are able to define activity diagrams. The theoretical approach to spread the diagram by text approach to other diagram types has been provided. It is not yet implemented and therefore several problems may still be unnoticed. The performance evaluation in section 3.5 has been completed. Its results show that the text based approach outperforms the drag & drop approach by the factor of two to five—depending on the complexity of the diagram. Another important result of this evaluation is that the hybrid text based approach does not outperform the drag & drop based one when drawing complex class diagrams. Therefore further research has to be done to make the text based approach even more competitive.

Goal (G7) has been reached. The existing custom element generation interface has been extended to enable the user to add the element to an existing text based diagram. The downside of this approach is that the complexity increases vastly. Therefore creating such elements is not suitable for novice users.

Goal (G8) provided a necessary basis for the implementation of other goals but has not been evaluated extensively yet.

Goal (G9) has been reached. The success can be easily measured by the amount of Eclipse features the application uses. Instead of only three—the drawing panel, the file creation wizard, and the saving mechanism—UMLet now also uses the Eclipse tool bar and the Eclipse Edit menu—instead of integrating the UMLet menu into the drawing panel.

7. Conclusion

7.1. Result Overview

The main goals: to achieve greater flexibility, efficiency, and usability have been reached by different approaches. Flexibility has been improved by allowing users to generate customized UML elements by Java code. Efficiency has been improved by implementing the possibility to generate whole diagrams by text. Finally the general usability of UMLet has been improved by adaption to todays software standards.

Although all improvements have been implemented partially within version 9.x of UMLet, some aspects have only have been covered theoretically. Therefore future work will include the implementation and enhancement of the topics presented in this thesis.

7.1.1. Improved Flexibility

Flexibility is a very important feature for almost every user. Therefore most tools provide some flexibility mechanism—from simple UI customization to plugin development. Usually the complexity of the extension mechanism increases along with the flexibility it provides. UI customization can be done by almost every user, but therefore gained flexibility is very limited. Plugins make an application very flexible, but plugin development is only suitable for experienced software developers.

UML adds other requirements to it's supporting tools. Because of the huge amount of UML elements and different versions of UML, a tool has to support a great amount of elements—new ones may be added every day. Therefore UML tools should provide a possibility to let the user design new elements.

UMLet achieves this by letting the end user design new elements by Java code. This provides the greatest flexibility in element design because the power of a whole programming language can be used. Another reason for choosing a programming language is that basic programming knowledge can be presumed, because UMLet is an UML drawing tool and UML diagrams are usually designed by software engineers.

Furthermore, UMLet implements several mechanisms to ensure that even beginner-programmers are able to design new elements:

The development environment is embedded into the application. This ensures that no media break occurs and no additional time is consumed by setting up the environment by hand. Furthermore UMLet uses templates to blend out code that is not needed to paint the element (e.g. the surrounding class's definition). By providing a set of basic methods that provide basic paint functions in an easy way, even beginners are able to create their first elements within minutes. Although the capabilities of these methods are limited, the flexibility of the approach is not decreased, because more experienced developers are able to create more complex elements by using Java's own libraries.

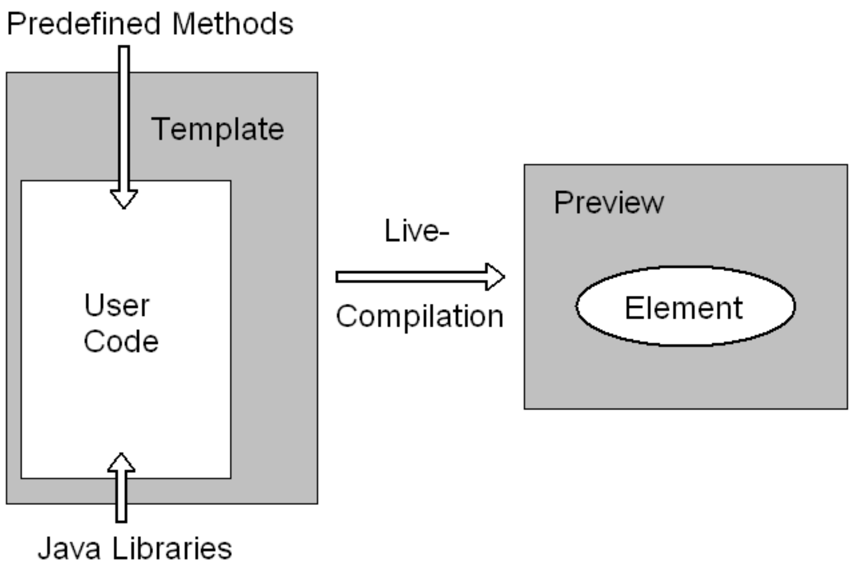


Figure 7.1.: Custom element generation

Another beneficial feature is that elements are compiled every few seconds automatically. If the compilation finishes without errors the element is then displayed immediately in a preview panel. This ensures that the user is confronted with the results of his code changes as they occur.

To sum up, UMLet provides a beginner friendly, very flexible, and quick responding mechanism for creating new diagram elements.

7.1.2. Improved Efficiency

Efficiency is very important for UML drawing applications—even more than to other UML tools, because no direct project benefits are gained—assuming that documentation purposes are no direct project benefits. By providing the possibility to create whole diagrams by entering text into a single field, UMLet provides a good alternative to generate diagrams at high speed. Within this thesis the *diagram by text* approach has been spread from sequence diagrams to activity diagrams. Activity diagrams have been chosen for adaption, because—like sequence diagrams—the positioning can be applied automatically in a predefined manner.

The gain of efficiency eventually did not come without a loss of flexibility. The implicit positioning prevents the user from placing the elements at special locations. Another disadvantage is that the diagram by text approach cannot be applied to other diagram types like class diagrams. There the positioning of elements is not predefined by the order the elements, but are placed using meta knowledge of the user (e.g. importance of the elements).

Therefore a hybrid approach has been introduced. This approach allows to create a general structure with basic elements via the text based approach. Then the user is able to adjust the diagram to his needs by using the usual drag&drop approach. This hybrid diagram creation has another advantage. Users do not need to know the textual grammar behind the diagram to start. They are able to create diagrams by drag&drop. The textual representation of the diagram is generated by the tool. Thus the user sees how he could have created this diagram by text and after some practice almost every user that favors the text based approach will be able to create diagrams by text.

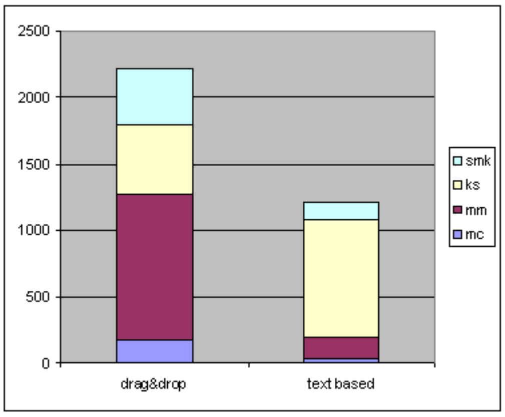


Figure 7.2.: Diagram by text - overall performance

The performance of creating diagrams by text usually outweighs diagram creation by drag&drop—

see figure 7.1.2. The approach performs better, if almost the whole diagram can be created using the textual grammar. As soon as many additional drag&drop operations are required to e.g. reposition the elements the approach looses some of the performance gain. Overall the approach fits best for diagrams that allow to position the elements automatically and do not require the user to rework the diagram after basic creation. This approach also increases performance slightly in the contrary situation. In the worst case the text based approach becomes another option of designing diagrams.

7.1.3. Improved Usability

Although already standard in almost every application the standard conformity and usability of UMLet has been improved vastly during this thesis. Starting with providing the possibility to edit multiple diagrams at once to adding copy/paste functionality. During development of the new features described in the sections above, UMLet's general usability has been updated to todays state of the art. Additionally UMLet's architecture has been refactored and the UMLet Eclipse plugin has been fully redesigned.

7.2. Summary: Lessons Learned

The main lessons that can be learned from this thesis are:

Provide immediate feedback

Providing immediate feedback to the user is a key feature when confronting him with complex tasks. The immediate feedback works well with both approaches: diagrams by text and custom element generation. The diagram by text approach refreshes the diagram with every keystroke. The custom element generation approach compiles the element periodically.

Hide needless information

Hide needless information from the end user. Although this rule applies to almost every area in software engineering it is especially beneficial when confronting the user with complex tasks. E.g. hiding the class body and the import statements from the end user has no effect on functionality, but halves the lines of code needed to generate a simple element. This provides a less complex introduction to a complex activity.

Avoid additional dialogs and external scripts

Functionality should be embedded into the application. This means that pop-up dialogs and external shell scripts shall be avoided. The user shall not be confronted with additional programs when using a more complex feature of a program (e.g. diagrams by text or custom element generation).

Creating diagrams by text is efficient

Using the diagrams by text approach is 2-5 times faster than creating diagrams by drag & drop. This makes diagram drawing very efficient which may encourage the users to use UML diagrams in early stages of software development.

Automatic element positioning in class diagrams is non trivial

Although very efficient, the diagram by text approach cannot be applied to every UML diagram type. As soon as an element's position depends on meta information—e.g. the importance of a class in a class diagram—automatic positioning algorithms cannot be applied. To spread the approach to other diagram types users have to be able to alter the diagram manually.

Merging the text based and the drag & drop based approach is efficient and flexible

Combining the text and drag & drop based approaches merges the advantages of both. The efficiency gained by the text based approach is merged with the flexibility of the drag & drop based one. Another benefit is that the text based approach may now be spread to other diagram types, because the user is able to alter the diagram after it has been generated.

Problems when combining the text based and the drag & drop based approach

The greatest problem when combining the two approaches is to avoid the loss of information. Due to the fact that the textual representation contains less information than the graphical one, information may be lost, when elements are removed and then reinserted in the textual representation (e.g. copy/paste operations).

Creating custom elements for text based diagrams increases complexity

Enabling the user to design custom elements for text based diagrams is much more complex than enabling the user to create them for the drag & drop based approach. If using the drag &

drop based approach, the user only has to implement a single drawing method. When creating elements for the text based approach he has to additionally implement mapping functions between the text and the graphical representation. This increases complexity and might discourage users from using this feature.

UMLet is a simple but powerful tool

UMLet is a simple tool for drawing UML diagrams. It also provides possibilities to create UML diagrams by text and enables the user to create customized elements.

7.3. Future Work

Future work on UMLet will include the implementation of the currently theoretical approach of hybrid diagrams—diagrams that can be created by text and/or drag&drop, whatever the user favors. After a first implementation the approach has to be spread to other diagram types to create a standardized interface for all diagrams.

After this is achieved the custom element generation presented in chapter 2 has to be updated to the one presented in chapter 4 to allow users to embed their newly designed custom elements into textual grammars.

Another open issue is to make exchanging custom elements easier. This could be achieved by providing an option to send custom elements to other users via email. An alternative approach is to provide an online repository where users can update, download, and rate custom elements.

Finally users could be enabled to design their own hybrid diagram grammars. This would make UMLet totally configurable by the user and would even allow it to be spread other types of diagrams than UML. This project is rather complex and it is more than doubtful that a solution would be simply enough to be suitable for common UMLet users.

A. Detailed Performance Analysis

A.1. Class Diagrams

N	Task description	mc	mm	ks	smk	E
1.	Move to right position in property panel	1	0	0	0	2
2.	Write text	0	0	154	1	160
3.	Reposition classes	0	6	0	1	24
4.	Reposition relations	0	12	0	0	36
	Overall	1	18	154	2	222

Table A.1.: Performance analysis - class - text based - simple

N	Task description	mc	mm	ks	smk	E
1.	Insert, position, and name Person	1	2	10	1	24
2.	Insert, position, and name Course	1	5	6	2	35
3.	Insert, position, and name Student	1	5	7	2	36
4.	Insert, position, and name Professor	1	5	9	2	38
5.	Insert, position, and name University	1	5	10	2	39
6.	Insert and position first inheritance	1	5	0	1	23
7.	Copy and position second inheritance	1	4	0	0	14
8.	Insert and position first association	2	6	7	1	35
9.	Copy and position second association	1	5	0	1	23
10.	Insert and position composition	2	6	7	1	35
11.	Insert and position relation	2	6	14	2	48
	Overall	14	54	70	15	350

Table A.2.: Performance analysis - class - drag & drop - simple

N	Task description	mc	mm	ks	smk	E
1.	Move to right position in property panel	1	0	0	0	2
2.	Write text	0	0	154	1	160
3.	Add details to Person	2	2	16	2	38
4.	Add details to Course	2	2	34	2	56
5.	Reposition and add details to Student	1	3	74	2	97
6.	Reposition and add details to Professor	1	3	13	2	36
7.	Reposition and add details to University	1	3	16	2	39
8.	Reposition inheritance	0	6	0	1	24
9.	Add info to attend relation	2	2	7	1	23
10.	Reposition and add info to lectures association	1	3	9	2	32
11.	Reposition and add info to employs association	1	5	8	2	37
12.	Add info to provides composition	2	2	9	2	31
	Overall	14	31	340	19	575

Table A.3.: Performance analysis - class - text based - complex

N	Task description	mc	mm	ks	smk	E
1.	Insert, position, and name Person (+details)	1	2	26	1	40
2.	Insert, position, and name Course (+details)	1	5	40	2	69
3.	Insert, position, and name Student (+details)	1	5	81	2	110
4.	Insert, position, and name Professor (+details)	1	5	22	2	51
5.	Insert, position, and name University (+details)	1	5	26	2	55
6.	Insert and position inheritance relations	2	9	0	1	37
7.	Insert, position, and name lectures association	2	6	16	1	44
8.	Copy, position, and name employs association	1	7	7	2	42
9.	Insert, position, and name provides composition	2	6	16	2	50
10.	Insert, position, and name attend relation	2	6	21	2	55
	Overall	14	56	255	17	553

Table A.4.: Performance analysis - class - drag & drop - complex

A.2. Activity Diagrams

N	Task description	mc	mm	ks	smk	E
1.	Insert and position start element	1	2	0	0	8
2.	Insert, position, and name first activity	1	5	4	1	27
3.	Insert and position first if element	1	3	0	1	17
4.	Insert, position, and name second activity	1	4	11	1	31
5.	Copy and position third activity from first	1	3	0	1	17
6.	Insert and position second if element	1	3	0	0	11
7.	Copy and position fourth activity from first	1	3	0	0	11
8.	Insert and position end element	1	3	0	0	11
9.	Insert connection 1	1	5	0	0	17
10.	Insert connection 2	1	4	0	0	14
11.	Insert and name connection 3	2	8	16	1	50
12.	Insert and name connection 4	2	8	7	2	47
13.	Insert connection 5	1	7	0	1	29
14.	Insert connection 6	1	7	0	0	23
15.	Insert connection 7	1	5	0	0	17
16.	Insert connection 8	1	4	0	0	14
	Overall	18	74	38	8	344

Table A.5.: Performance analysis - activity - drag & drop - simple

N	Task description	mc	mm	ks	smk	E
1.	Insert all-in-one element	1	0	0	0	2
2.	Move to right position in property panel	1	1	0	0	5
3.	Write text	0	0	57	1	63
	Overall	2	1	57	1	70

Table A.6.: Performance analysis - activity - text only - simple

N	Task description	mc	mm	ks	smk	E
1.	Move to right position in property panel	1	0	0	0	2
2.	Write text	0	0	57	1	63
	Overall	1	0	57	1	65

Table A.7.: Performance analysis - activity - hybrid - simple

N	Task description	mc	mm	ks	smk	E
1.	Insert all-in-one element	1	0	0	0	2
2.	Move to right position in property panel	1	1	0	0	5
3.	Write text	0	0	339	1	345
	Overall	2	1	339	1	352

Table A.8.: Performance analysis - activity - text only - complex

N	Task description	mc	mm	ks	smk	E
1.	Move to right position in property panel	1	0	0	0	2
2.	Write text	0	0	328	1	339
3.	Move connection to document call	0	4	0	1	18
	Overall	1	4	328	2	354

Table A.9.: Performance analysis - activity - hybrid - complex

N	Task description	mc	mm	ks	smk	E
1.	Insert and position start element	1	2	0	0	8
2.	Insert, position, and name tel. rings activity	1	5	4	1	27
3.	Insert and position middle if element	1	3	0	1	17
4.	Insert and position fork bar	1	3	0	0	11
5.	Insert and position left if element	1	3	0	0	11
6.	Insert, position, and name take call activity	1	5	9	1	32
7.	Insert, position, and name help caller activity	1	5	11	2	40
8.	Insert, position, and name forward call activity	1	5	12	2	41
9.	Insert and position left end if element	1	3	0	1	17
10.	Insert, position, and name document call activity	1	5	13	1	36
11.	Insert and position sync element	1	3	0	1	17
12.	Insert and position middle end if element	1	3	0	0	11
13.	Insert and position end element	1	3	0	0	11
14.	Insert, position, and name record call activity	1	5	11	1	34
15.	Insert, position, and name send event	1	5	15	2	44
16.	Insert, position, and name receive event	1	5	4	2	33
17.	Insert, position, and name playback	1	5	22	2	51
18.	Insert connection 1	1	5	0	1	23
19.	Insert connection 2	1	4	0	0	14
20.	Insert and name [free] connection	2	5	9	1	34
21.	Insert fork connection 1	1	7	0	1	29
22.	Insert fork connection 2	1	6	0	0	20
23.	Insert and name [responsible] connection	2	7	16	1	47
24.	Copy and change name of [not resp.] connection	2	8	4	2	44
25.	Insert left if block connection 1	1	5	0	1	23
26.	Insert left if block connection 2	1	6	0	0	20
27.	Insert left if block connection 3	1	6	0	0	20
28.	Insert sync connection 1	1	6	0	0	20
29.	Insert sync connection 2	1	6	0	0	20
30.	Insert sync connection 3	1	5	0	0	17
31.	Insert and name [no open calls] connection	2	5	18	1	43
32.	Copy and change name of [open calls] connection	1	11	1	2	48
33.	Insert and name [busy] connection	2	8	9	2	49
34.	Insert send event connection	1	5	0	1	23
35.	Insert receive event connection	1	4	0	0	14
36.	Insert connection to document call	1	6	0	0	20
	Overall	41	183	158	30	969

Table A.10.: Performance analysis - activity - drag & drop - complex

B. Bibliography

[1] M. Auer, L. Meyer, and S. Biffl. Explorative UML modeling: Comparing the usability of UML tools. In *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS'07)*, pages 466–474, Madeira, 2007.

[2] M. Auer, T. Tschurtschenthaler, and S. Biffl. A flyweight UML modeling tool for software development in heterogeneous environments. In *Proceedings of the 29th EUROMICRO Conference*, pages 267–272, Antalya, 2003.

[3] J. Beringer. Reducing expertise tension. *Communications of the ACM*, 47(9):39–40, 2004.

[4] J. Beringer, G. Fischer, P. Mussio, B. Myers, F. Paternò, and B. de Ruyter. The next challenge: From easy-to-use to easy-to-develop. Are you ready? In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'08)*, pages 2257–2260, Florence, 2008.

[5] S. Berti, F. Paternò, and C. Santoro. Natural development of nomadic interfaces based on conceptual descriptions. In *End-User Development*, pages 143–160. Springer, 2006.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 2. edition, 2005.

[7] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[8] W. Cazzola, R. Stroud, and F. Tisato. *Reflection and Software Engineering*. Springer, 2000.

[9] P. Coad and E. Yourdon. *Object-Oriented Design*. Prentice-Hall, 1991.

[10] M. Costabile, P. Mussio, L. Provenza, and A. Piccinno. End users as unwitting software developers. In *Proceedings of the 4th International Workshop on End-User Software Engineering (WEUSE'08)*, pages 6–10, Leipzig, 2008.

[11] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, Anaheim, 2003.

[12] M. Eiglsperger, M. Kaufmann, and M. Siebenhaller. A topology-shape-metrics approach for the automatic layout of UML class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 189–ff, San Diego, 2003.

[13] A. Evans and A. Wellings. UML and the formal development of safety-critical real-time systems. In *IEE Colloquium on Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems*, pages 2/1–2/4. 1999.

[14] G. Fischer, E. Giaccardi, Y. Ye, A. Sutcliffe, and N. Mehandjiev. Meta-design: A manifesto for end-user development. *Communications of the ACM*, 47(9):33–37, 2004.

[15] G. Fischer and E. Scharff. Meta-design: Design for designers. In *Proceedings of the 3rd Conference on Designing Interactive Systems (DIS'00)*, pages 396–405, New York, 2000.

[16] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[17] M. Heng. Beyond end user computing. In *Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS'03)*, pages 594–598, Angers, 2003.

[18] P. Huber. The model transformation language jungle - an evaluation and extension of existing approaches. Master's thesis, Vienna University of Technology, 2008.

[19] ISO/IEC. ISO/IEC 14977: Information technology - Syntactic metalanguage - Extended BNF. Technical report, International Organization for Standardization, Geneva, Switzerland, 1996.

[20] A. Ko and B. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing*, 16(1-2):41–84, 2005.

[21] H. Kohler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML diagrams for production control systems. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 241–251, Limerick, 2000.

[22] S. Lahtinen and J. Peltonen. Enhancing usability of UML CASE-tools with speech recognition. In *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments (HCC'03)*, pages 227–235, Auckland, 2003.

[23] J. Ludewig. Models in software engineering – an introduction. *Software and Systems Modeling*, 2(1):5–14, 2003.

[24] J. Martin. *An Information Systems Manifesto*. Prentice Hall, 1. edition, 1984.

[25] J. McConnel. Active learning and its use in computer science. *ACM SIGCSE Bulletin*, 28(SI):52–54, 1996.

[26] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 37(4):316–344, 2005.

[27] J. Nielsen. *Usability Engineering*. Academic Press, 1993.

[28] B. de Ruyter and R. van de Sluis. Challenges for end-user development in intelligent environments. In *End-User Development*, pages 243–250. Springer, 2006.

[29] J. Segal. Some problems of professional end-user developers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'07)*, pages 111–118, Coeur d'Alène, 2007.

[30] H. Smith. On tool selection for illustrating the use of UML in system development. *Journal of Computing Sciences in Colleges*, 19(5):53–63, 2004.

[31] D. Spinellis. On the declarative specification of models. *IEEE Software*, 20(2):94–96, 2003.

[32] A. Sutcliffe. Evaluating the costs and benefits of end-user development. *SIGSOFT Software Engineering Notes*, 30(4):1–4, 2005.

[33] D. Tamir, O. Komogortsev, and C. Mueller. An effort and time based measure of usability. In *Proceedings of the 6th International Workshop on Software Quality (WoSQ)*, pages 47–52, Leipzig, 2008.

[34] Object management group (OMG), 2008. http://www.omg.org.

[35] Unified Modeling Language (UML), version 2.1.2, 2008. http://www.omg.org/technology/documents/formal/uml.htm.

[36] Vinita, A. Jain, and D. K. Tayal. On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms. *ACM SIGSOFT Software Engineering Notes*, 33(5), 2008.

[37] E. Yourdon and L. Constantine. *Structured Design*. Prentice-Hall, 1979.