FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Lightweight UML Model Creation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Ludwig Meyer
Matrikelnummer 9656711

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Mag.rer.soc.oec. Stefan Biffl
Mitwirkung: Dipl.-Ing. Mag. Dr.techn. Martin Auer, M/A

Wien, 14.03.2011 ________________________ ________________________

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Abstract:

The UML is an industry-wide accepted modeling standard that offers 13 diagram types using simple notation. Although there is broad tool support many of these tools are hard to use. They suffer from non-optimal designed user interfaces that rely on hard decipherable icons, use of pop-up windows, nested menus and tab-panes, etc. An alternative to plain graphical metaphors is text-based user input. The open-source tool UMLet integrates the text-based approach into modeling by creating elements and even entire diagrams by the use of simple grammar.

This work compares the efficiency of traditional graphical user interfaces and UMLet's text-based approach in the context of explorative UML modeling using a set of representative use-cases that are essential to UML modeling. The evaluation features a simplified method for quantitative measuring of user interfaces focused on the amount of user interactions needed to fulfill a task.

UMLet is a lightweight UML sketching tool. Its development is focused on a simple user interface. Fast diagram creation at the speed of paper modeling but with the advantage of more flexibility when altering the design.

UMLet's architecture is design-pattern based. Since it is provided as an open source tool, well known source code constructs assist and ease the contribution of source code by users.

An unobtrusive user interface is achieved by avoiding pop-up windows. The information used to specify the entities is not spread across several windows but can be entered at a single place. The text-based user interface increases interaction speed when creating diagrams. The modifications are instantly assigned to the selected element where the changes take effect just in time. UMLet even provides entirely grammar empowered diagram types. Sequence diagrams, activity diagrams, and a number of smaller elements can be built using a simple yet powerful syntax.

A key feature of UMLet is its expandable modular structure accomplished with a built-in Java compiler. Users may implement their custom elements on the fly using Java. The tool provides a simple template element explaining all necessary components that can be easily edited. Sharing the self-created elements is as easy as providing the Java source code files.

Support for a broad variety of notations is provided by the above-mentioned custom elements feature and the fact that UMLet comes with a nearly full-featured UML support.

Kurzfassung:

UML ist ein industrieweit anerkannter Modellieungs-Standard mit 13 Diagrammtypen und einfacher Notation. Obwohl es breite Tool-Unterstützung gibt, sind viele dieser Tools nur schwer bedienbar. Sie leiden unter nicht optimal entworfenen Benutzeroberflächen mit schwer entzifferbaren Icons, Pop-Up Fenstern, verschachtelten Menüs und Tab-Panes, etc. Text-basierte Benutzereingabe bietet eine Alternative zu bekannten grafischen Metaphern. Das open-source Tool UMLet integriert den text-basierten Ansatz beim Modellieren, um Elemente und Diagramme mittels einfacher Grammatik zu erstellen.

Diese Arbeit vergleicht die Effizienz herkömmlicher GUIs mit UMLets text-basiertem Interface im Kontext des explorativen UML Modellierens auf Basis repräsentativer Use-Cases. Die Evaluierung beruht auf einer vereinfachten Methode zur quantifizierten Messung von Benutzeroberflächen. Sie betrachtet die Anzahl der durch den Nutzer erforderlichen Interaktionen zur Erfüllung einer Aufgabe.

UMLet ist ein leichtgewichtiges Werkzeug zur UML Modellierung. Der Fokus liegt auf einem einfachen und verständlichen Bedienungskonzept. Die Erstellung von Diagrammen so einfach und schnell wie mit Papier und Bleistift, aber mit den Vorteilen bei der Modifikation und Verfeinerung am Bildschirm.

Da UMLet ein quelloffenes Projekt ist, basiert die Architektur auf Design-Patterns um die Verständlichkeit zur erhöhen und Beiträge aus der Community zu fördern.

Das unaufdringliche Bedienungskonzept entsteht unter anderem durch den Verzicht auf Pop-Up Fenster. Die Information zur Spezifizierung von Entitäten ist nicht über mehrere Fenster verteilt, sondern an einem einzelnen Ort konzentriert. Ein text-basiertes Interface erhöht die Bedienungsgeschwindigkeit bei der Erstellung von Diagrammen. Einfache aber zugleich mächtige Syntax erlaubt die Modifikation einzelner Elemente, und sogar ganzer Diagrammtypen, wie beispielsweise Sequenzdiagramme oder Activity-Diagramme.

Eines der Schlüsselfeatures UMLets ist seine erweiterbare, modulare Struktur mithilfe eines integrierten Java-Compilers. Die Benutzer können anhand eines einfachen Templates, das bereits alle notwendige Basisfunktionalität aufweist, einfach und schnell mittels Java ihre eigenen Elemente erstellen. Diese selbst erstellten Elemente können auch einfach an weitere Benutzer verteilt werden, da sie einfache Java-Quelldateien sind.

UMLet bietet eine fast komplette Unterstützung für UML 2.0 und ist nicht zuletzt durch die Erweiterbarkeit mit Hilfe der genannten Custom-Elements an die jeweiligen Anforderungen leicht anpassbar.

Eidesstattliche Erklärung:

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, 14.03.2011

Ludwig Meyer

Contents

Chapter 1

Introduction

Models are essential to the understanding of systems and processes. The UML is a widely accepted modeling standard offering 13 diagram types. Although the UML offers simple notation many tools are difficult to use. They rely on conventional user interface design with hard decipherable icons, use of modal dialog pop-ups, nested menus and tab-panes, etc.

Text-based input bears an alternative to plain graphical metaphors. UMLet integrates the text-based approach into UML modeling by creating elements and even entire diagrams by the use of simple grammar. This work compares the efficiency of traditional graphical user interfaces and UMLet's text-based approach in the context of explorative UML sketching.

The research hypothesis is that *text-based user input can reduce the number of user interactions with a graphical user interface, increase the speed of user input and lead to a more efficient user interface.* Chapter 6 explains the motivation, presents the context and setup of the evaluation to prove the hypothesis.

UML and design patterns

The Unified Modeling Language (UML) has become an accepted standard notation in software engineering. UML is standardized and further developed by the UML Partners consortium formed by well-known companies. It provides a graphical notation and at the moment it features 13 different diagram types suitable for most phases of the design process of software development from requirements engineering to deployment.

Chapter two is divided into two parts. Part one gives an introduction to visual modeling with the UML, its diagrams, and elements. Part two gives an overview of common design patterns.
In the development of UMLet design patterns play a significant role. Since the tool is open source all users are invited to modify UMLet to match their spe-

cial needs and contribute their ideas to the project. To help people understand UMLet's structure and internals, design patterns are used where applicable and reasonable.

The chapter's part on design patterns introduces the so called GOF patterns as they are utilized in UMLet. The other GOF patterns and the concept of anti patterns are introduced shortly as well.

Requirements for lightweight modeling

Chapter three discusses the requirements of lightweight UML tools:

1. Easy to use: A well designed user interface is essential for all software applications made for direct communication with humans.

2. Easy to learn: The design of the software and its workflows should be easily understandable and usable.

3. Easy to deploy: When deploying a software tool to more than one workstation and operating system, the process of software installation may be critical. Using a software tool for educational processes requires simple installation processes, especially when students are required to install the software on their home computers.

4. Relaxed standards restrictions: Explorative sketching is the process of creating not necessarily exact and complete diagrams. Tools that enforce strict standards conformity slow down the design process and draw too much attention to the software tool itself, than to the creation and development progress.

Fowler [20] expresses his opinion about explorative sketching and strict and detailed model creation:

> "Almost all the time, my use of the UML is as sketches. I find the UML sketches useful with forward and reverse engineering and in both conceptual and software perspectives. I'm not a fan of detailed forward-engineered blueprints; I believe that it's too difficult to do well and slows down a development effort. Blueprinting to a level of subsystem interfaces is reasonable, but even then you should expect to change those interfaces as developers implement the interactions across the interface. The value of reverse-engineered blueprints is dependent on how the tool works. If it's used as a dynamic browser, it can be very helpful; if it generates a large document, all it does is kill trees." [20, page 6]

Related work

Chapter 4 presents related work on the UML, the usage of UML, and UML tools. It furthermore presents relevant work on alternative user interfaces as well as suggestions to good user interface design, and the assessment of user interfaces. Raskin's [36] work on quantitative measuring methods on user interfaces is the basis of chapter 6's evaluation. An outline of UMLet's direct competitors completes this chapter.

Lightweight tools

An article on ZDNet.com summarizes some e-mails and memos of top Microsoft executive Bill Gates and top level employee Ray Ozzie about the future challenges of the internet and discusses their competitors like Adobe, Google, Skype and others. In one of the key statements Microsoft's Chief Software Architect Ray Ozzie points out the need of simplicity of software tools:

> "'Developers needing tools and libraries to do their work just search the Internet, download, develop and integrate, deploy, refine,' Ozzie wrote. 'Speed, simplicity and loose coupling are paramount.' "
> ['Gates memo warns of 'disruptive' changes', 2005 ZDNet][1]

UMLet is such a lightweight UML sketching tool. It has a simple user interface avoiding pop-ups. The simplicity of its user interface makes it easy to use and easy to learn, as stated above.
It features a simple standard file format using XML to ease the integration into other tools. A simple XSL transformation might even generate source code stubs from UMLet diagrams.

Having been developed using pure Java UMLet is independent from operating systems and platforms. It is possible to deploy UMLet to Windows, Unix/Linux, Apple, and all other platforms that have a Java runtime environment. It does not use any platform-specific features or need any configuration settings.
UMLet may be used as a stand alone Java application and furthermore can be integrated into the Eclipse software development environment as a plug-in. It uses a transparent way of drawing to the display device making the development independent from the actual output device.

Customizable palettes simplify UMLet's usage. Rather than using tiny icons UMLet's palettes hold the graphical elements in their real size. This feature allows the user to identify the elements directly because the palettes do not only show the elements in their real size, but also their different occurrences and example usages. Custom elements allow flexible element creation using Java featuring a compiler at runtime. These custom elements may be shared easily

[1]http://www.zdnet.com.au/gates-memo-warns-of-disruptive-changes-139221468.htm

by distributing the Java source files.

Making UMLet interoperable with common publishing software, it offers to export the diagrams to a variety of wide spread formats like EPS, PDF and others.

Chapter 5 discusses UMLet's design goals and solution approaches.

Educational use

Using software tools in education requires not only adequate licensing models but there are some requirements that are crucial for a successful course. These requirements cover issues like support for different publishing formats to import the diagrams into text processing applications or publish them on web pages.

A critical issue is the simplicity of the tool's user interfaces. Especially in education, software tools need to have simple user interfaces to allow the students to concentrate on solving their exercises and not bother by complicated software.

Another question is the possibility to expand the tool to match the requirements of the course. UML tools often lag behind UML's standardization process, providing only a subset of the standard in force. UMLet offers the users the possibility to add their custom elements and diagram types.

The heterogeneous computer system environments as they can be found in schools and universities require the software tools to utilize simple ways of integration. The development of UMLet also addresses this issue requiring minimal user intervention.

Tool comparison

Most UML tools use windows and pop-up dialogs with more or less difficult user interfaces to modify the characteristics of their elements. UMLet approaches this issue by introducing a text-based modification method using a single centralized attributes window. The elements feature a simple syntax. There are even entire diagram types that may be created using a simple yet powerful grammar.
Chapter 6 focuses on the comparison of UMLet and the market-leading tool in software engineering Rational Rose. The comparison is based on a set of 16 representative use cases covering basic tasks and the more sophisticated challenges of creating UML diagrams. Both tools are evaluated and compared using a simplified method for quantitative measuring of user interface design.

Results

Chapter 7 presents a summary and discussion of the evaluation's results. It furthermore takes a look at the development progress of UML tools, takes a critical look at UMLet and its present problems and presents feedback and experiences of users of UMLet.
Finally chapter 8 gives an overall summary and motivation for future work.

Note: Parts of this work have been previously published in this paper [4].

Chapter 2

UML and Design Patterns

2.1 Visual modeling with the UML

> "Graphical design notations have been with us for a while. For me, the primary value is in communication and understanding. A good diagram can often help communicate ideas about a design, particularly when you want to avoid a lot of details. Diagrams can also help you understand either a software system or a business process. As part of a team trying to figure out something, diagrams both help understanding and communicate that understanding throughout a team. Although they aren't, at least yet, a replacement for textual programming languages, they are a helpful assistant." [20, page XXVI]

There are a number of people proposing different notations for visual modeling. The notations with the actual strongest support are Booch, OMT (Object Modeling Technology) and UML.

1. The Booch Method: Grady Booch, the inventor of the graphical notation named after him, has written several books on the benefits of visual modeling. At the time of developing this notation he worked at Rational Software Corporation. This notation utilizes clouds for representing objects and a set of arrows for representing the relationships between these objects.

2. Object Modeling Technology: OMT was developed by another famous person in the field of software engineering: Dr. James Rumbaugh. OMT uses simpler graphical objects than Booch for illustrating.

3. The Unified Modeling Language (UML): The UML notation was born by the collaboration of Grady Booch, Dr. James Rumbaugh, Ivar Jacobson working at Rational Software Corporation and others. While the UML

elements closely match those of Booch and OMT, UML integrates elements from further notations.

> "The consolidation of these three methods, that became the UML, started in 1993. Each of the three amigos of UML began to incorporate ideas from the other methodologies. Official unification of the methodologies continued until late 1995, when version 0.8 of the Unified Method was introduced. The Unified Method was refined and changed to the Unified Modeling Language in 1996. UML 1.0 was ratified and given to the Object Technology Group in 1997, and many major software development companies began adopting it. In 1997, OMG released UML 1.1 as an industry standard. Over the past years, UML has evolved to incorporate new ideas such as web-based systems and data modeling." [8, page 12]

In 1996 the UML Partners consortium was formed by well known companies such as IBM, Microsoft, HP, Digital Equipment, Oracle and many others participating in the further development and allowing the UML to develop independently of any of the companies to form an industry-wide accepted standard.

A big advantage of the UML is the fact that it is available to everyone for free without the hassle of fees and licenses. It is backed by a number of tools and available books that were written without any fear of patent or license struggles.

UML is the acronym for Unified Modeling Language.

Unified: A common language is the base for communication. The Object Management Group (OMG) and Rational Software Corporation (which was meanwhile acquired by IBM in 2003), who initially created the UML, had to merge the best and most accepted techniques and engineering practices, bring the standards together and unify the approaches and procedures (see below).

Rumbaugh et. al. [38, page 9] state that the term unified also refers to the unification across:

- historical methods and concepts,
- the development cycle meaning that the same concepts and notations may be seamlessly utilized within the different stages of development without the need for transformation from one technique to another,
- application domains stating that the UML is applicable for large scale systems, real-time applications and other more or less specialized systems as well
- implementation languages and platforms being independent from any specific programming language and runtime or development environments

- the development process intended to be the underlying modeling language supporting an iterative and incremental development process
- the internal concepts providing seamless and coherent modeling concepts easing the application of them on well-known and unknown situations.

Model: A model is a simplified representation of reality. The simplification is done by applying valid abstractions. Models are used to illustrate a part of the real world while eliminating unnecessary or too complex details. By managing the used abstractions, models help handling the complexity of problems. A definition:

> "An approximation, representation or idealization of selected aspects of the structure, behavior, operation, or other characteristics of a real-world process, concept, or system. Note: models may have other models as components." [27, page 133]

> "[...] a useful technique is to abstract the system into a simplified model, removing the minutia of the system in order to have a more understandable version. The purpose of modeling is to simplify the details down to an understandable "essence" but to not oversimplify to the point that the model does not adequately represent the real system. In this way, we can think about the system without being buried in the details." [30, page 295]

Models combine semantics and presentation.

> "The semantic aspect captures the meaning of the application as a network of logical constructs, such as classes, associations, states, use cases, and messages. The semantic model elements carry the meaning of the model [...]. The semantic information is often called *the model*. A semantic model has a syntactic structure, well formedness rules, and execution dynamics. These parts are often described separately (as in the UML definition models), but they are tightly interrelated parts of a single coherent model." [38, page 19]

> "The visual presentation shows semantic information in a form that can be seen, browsed, and edited by humans. Presentation elements carry the visual presentation of the model – that is, they show it in a form directly apprehensible by humans. They do not add meaning, but they do organize the presentation to emphasize the arrangement of the model in a usable way. They therefore guide human understanding of a model." [38, page 20]

Language: Without any form of language, it would not be possible to communicate at all. The key to successful communication is that all members who want to communicate with each other, agree to a set of symbols and the set of rules to qualify the semantics: the language. A language does not fundamentally have to be composed of written or spoken words. UML uses abstract symbols like rectangles to define the limits of an entity like objects and arrows to represent messages or relations between the participants of the system.

2.1.1 Modeling with the UML

The UML is based on the object-oriented approach and therefore it is not only focused on software engineering problems but it is possible to describe problems occurring in the subject areas of the real world business like finance and economics. The sequence diagram is not only capable of showing the life-cycle of a set of object instances in a piece of software, but it's potential covers describing complex financial transactions or stock exchange processes.

> "The Unified Modeling Language (UML) is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system. It captures decisions and understanding about systems that must be constructed. It is used to understand, design, browse, configure, maintain, and control information about such systems. It is intended for use with all development methods, lifecycle stages, application domains, and media. The modeling language is intended to unify past experience about modeling techniques and to incorporate current software best practices into a standard approach." [38, page 3]

The object-oriented approach in software design provides solutions to model entities (objects) and their relationships that may represent complex processes. By applying these techniques to business modeling allows speaking in the same language using the same terms. Achieving this allows using the same names in the software design and the real world.

> "For example, a "thing," such as a payroll withholding stub, described in the business domain might relate to a "thing" that appears again in the software domain – a payroll withholding record, for example. If we can be fortunate enough to use the same techniques or very similar techniques for both problem analysis and solution design, the two activities can share these same work products." [30, page 61]

> "One of the goals of business models is to develop a model of the business that can be used to drive application development." [30, page 62]

> "The use of object-oriented modeling for the purpose of business engineering has generated a lot of interest. Object-oriented models have proven to be an excellent method for modeling the business processes in a company. A business process provides some value to the customer of the business (or perhaps the customer's customer).

> When a company uses techniques such as Business Process Reengineering (BPR) or Total Quality Management (TQM), the processes are analyzed, improved, and implemented in the company. Using an object-oriented modeling language to model and document the processes also makes it easier to use these models when building the information systems in the company." [17, page 13]

UML itself is defined by a meta-model. A meta-model is a stereotype of a model. It expresses how to describe models. The meta-model of the UML describes the structure of UML models. The exact knowledge of UML's meta-model is only necessary if it is important to have exact models. For example if the diagram is used for source code creation or other UML-based processing like model verification.

> "How much does the meta-model affect a user of the modeling language? The answer depends mostly on the mode of usage. A sketcher usually doesn't care too much; a blueprinter should care rather more. It's vitally important to those who use the UML as a programming language, as it defines the abstract syntax of that language." [20, page 9]

2.1.2 Diagrams

UML features 13 official classified diagram types that are grouped together into diagrams that describe structural characteristics, behavioral description and interaction specification, while the latter is a subgroup of the behavioral diagrams.

> "[...] the UML's authors do not see diagrams as the central part of the UML. As a result, the diagram types are not particularly rigid. Often, you can legally use elements from one diagram type on another diagram. The UML standard indicates that certain elements are typically drawn on certain diagram types, but this is not a prescription." [20, page 11]

The by the UML 2 standard defined diagrams are:

1. Class Diagram
2. Component Diagram
3. Composite Structure Diagram
4. Deployment Diagram
5. Object Diagram
6. Package Diagram

7. Activity Diagram
8. Use case Diagram
9. State Machine Diagram
10. Sequence Diagram
11. Communication Diagram
12. Interaction overview
13. Timing Diagram

Class diagram

The nature of class diagrams is to describe the classes of a system and the various static relationships among them.

A *class* describes things of a certain type. These things may be classified and ordered within a generalization hierarchy represented by inheritance. Like the `Worker` presented in figure 2.2 is a specialized kind of `Person`. Classes describe groups of related objects having a state and behavior. The states are represented by attributes and associations subsumed by the term *property*.

> "A class is the descriptor for a set of objects with similar structure, behavior, and relationships. All attributes and operations are attached to classes or other classifiers." [38, page 50]

The basic element of a class a diagram is the class. It may occur in different flavours from just having a name to a full blown representation with attributes and operations (Figure 2.3).

Classes may show details with their properties and operations and their visibility constraints. Properties and operations may be specified with their visibility marks: '+', '-' and '#'. While the plus represents global visibility and the minus an invisibility outside the objects borders, the hashing symbol represents Java's *protected* state (this is not part of the UML2 standard, but in common use). So-called *class variables* that are uniquely shared within all instances are underlined.
Figure 2.3 also shows how to specify data types of variables and return parameters.

Properties represent the variables of a class. There are two ways of specifying the properties of classes:

1. *Attributes* can have additional visibility marks and additional type descriptors.

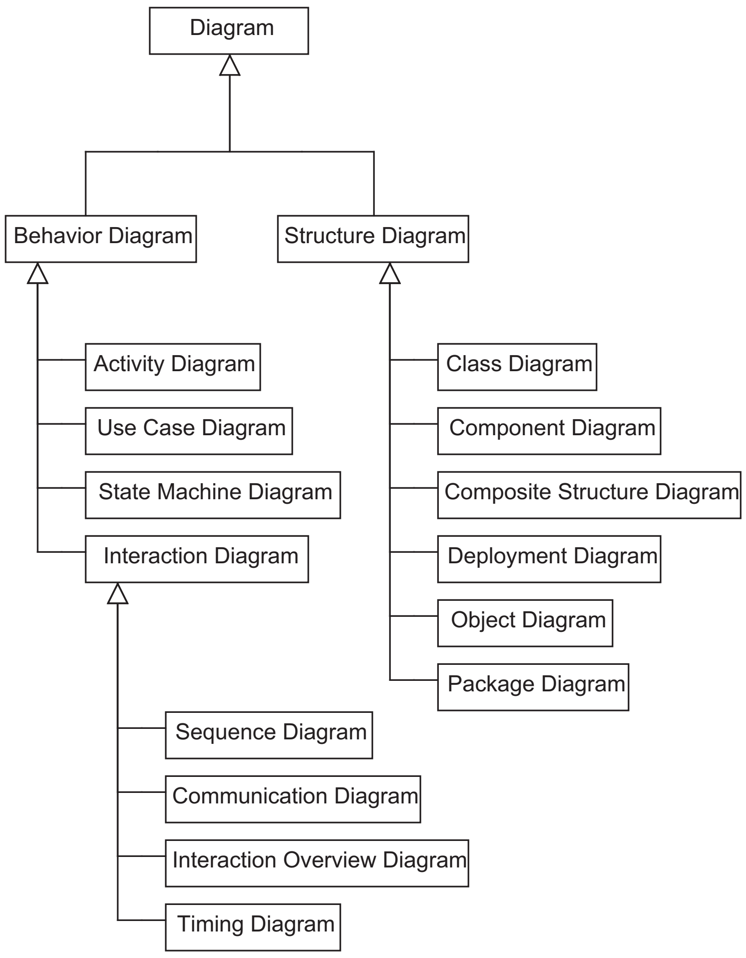


Figure 2.1: UML's diagram classification following [20, page XXVI]

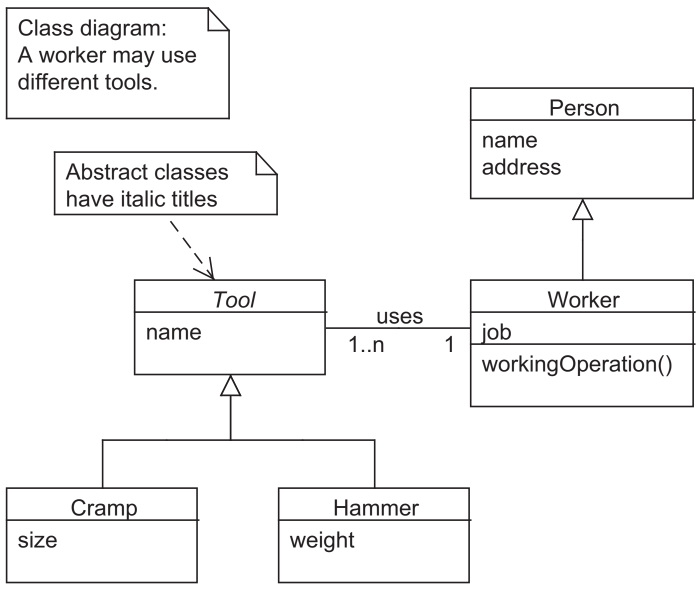


Figure 2.2: A class diagram featuring inheritance and relationships

2. *Associations* are represented by solid lines connecting the elements and its name. Directions may be shown by small filled arrows near the name, or by an open arrow head at the destination. Another way to specify an association is to attach an *association class* to it. Figure 2.4 shows three examples.

Qualified associations have an extension at the end of the arrow forming a box with the qualification. The qualification is used with one–to–many or many–to–many associations. It qualifies each single association. See figure 2.5

> “A qualified association is the UML equivalent of a programming concept variously known as associative arrays, maps, hashes, and dictionaries.” [20, page 74]

Aggregations and *compositions* are special types of associations. In contrast to the composite structure diagram that describes a set of collaborating elements the aggregation/composition describes the composition of objects forming a superior system. The difference between the aggregation and composition is that the composition represents strong ownership — it *owns* its parts. If the object formed by the composition is not instantiated yet, the parts are not instantiated too and the same behavior applies for destroying the composite: all parts are destroyed too.

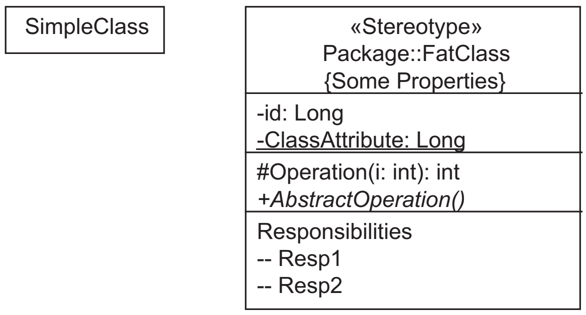


Figure 2.3: Basic versus full blown class representation

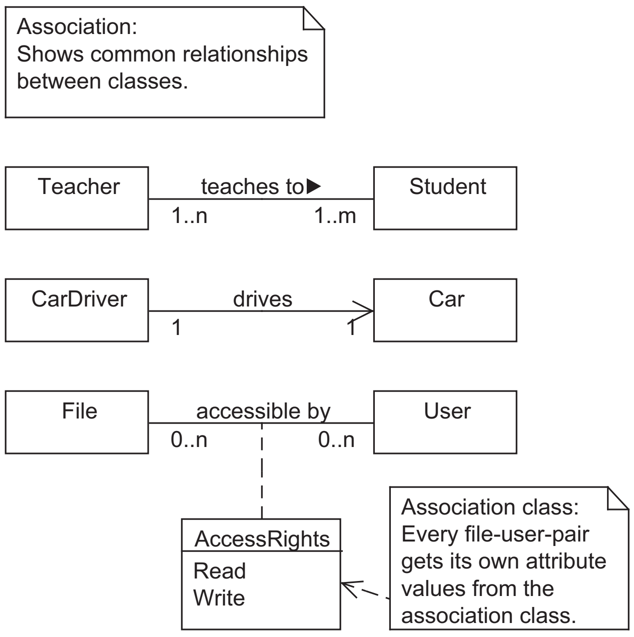


Figure 2.4: Associations

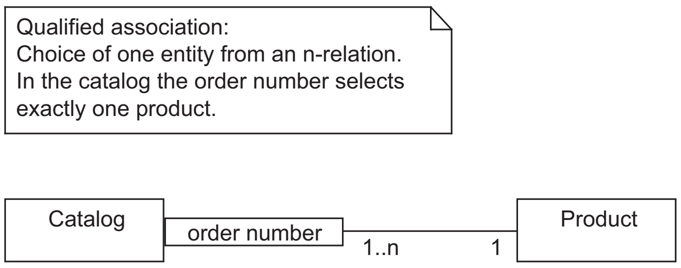

Qualified association:
Choice of one entity from an n-relation.
In the catalog the order number selects exactly one product.
Catalog
order number
1..n
1
Product

Figure 2.5: Qualified association

> "Aggregation is a special case of association. The aggregate indicates that the relationship between the classes is some sort of 'whole-part'. One example of an aggregate is a car that consists of four wheels, an engine, a chassis, a gear box, and so on. [...] An aggregation represents a strict relationship indicating the composition of a class, not the runtime architecture." [17, page 111]

> "A composition is a stronger form of association in which the composite has sole responsibility for managing its parts, such as their allocation and de–allocation." [38, page 56]

Aggregation and composition are graphically distinguished by the type of diamond they have. The aggregation owns a hollow diamond at the side of the composite (Figure 2.6), while the composition is drawn with a filled diamond (Figure 2.7).

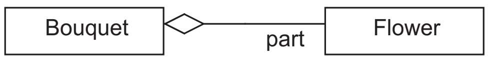


Figure 2.6: Aggregation

The *multiplicity* is shown on both ends of the association and describes how many participants are required or may be engaged. If an association has no multiplicity specified a value of one is taken by default.

- 1 requires exactly one object
- 0..1 allows zero or one occurrence
- * describes zero or an unbound number

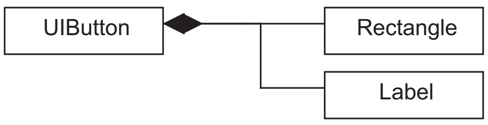


Figure 2.7: Composition

- it is also possible to define an upper bound and a lower bound like 3..5

UMLet's textual description for relations covering inheritance, associations, and other dependencies notated by arrows is as follows:

```
lt=<.>
m1=1
m2=2
r1=Role 1
r2=Role 2
q1=qualification 1
q2=qualification 2
Name
```

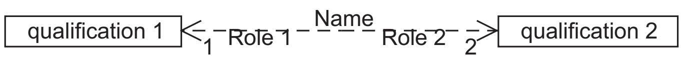


Figure 2.8: Relations in UMLet

While `lt` specifies the line type and form of the arrow heads, `m1` and `m2` specify multiplicities, `r1` and `r2` specify the roles, `q1` and `q2` name the qualifications, and the un-noted string `name` names the association. So UMLet allows one to modify the type and appearance of a relation in a very simple way. The given example is depicted in figure 2.8.

Generalization, Specialization and Inheritance: Generalization is the process of abstraction from a more specified item to a more general item. Like the term 'house' is a generalization of 'hut' and 'sky scraper', or 'animal' is a generalization for 'bird', and 'fish' but not for 'tree'. Specialization is the reverse process of generalization and specifies a particular item with more details. The object oriented approach utilizes inheritance to model generalization and specialization. The class diagram uses a special type of relation to express generalization. The arrow is headed towards the more general class. See figure 2.9.

Some programming languages like C++ or Eiffel allow multiple inheritance, which means that a class has more than one superclass it inherits attributes and methods from. Java prohibits multiple inheritance from classes but offers a compromise by allowing inheritance from one superclass while allowing one to implement additional interfaces. See figure 2.10.

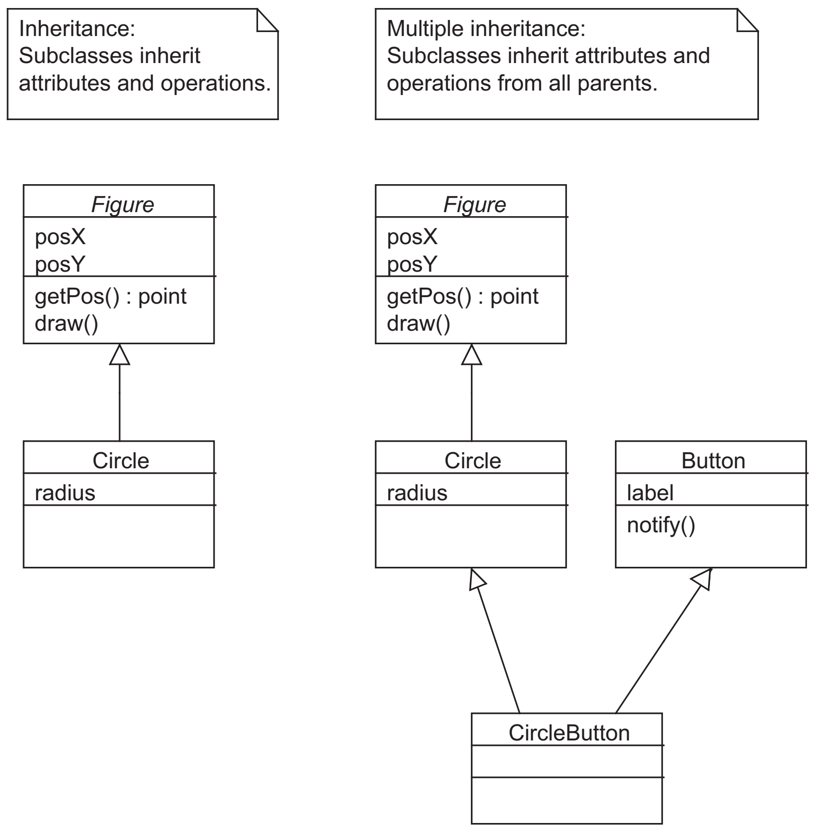


Figure 2.9: Inheritance

Dependency: A dependency describes a relation between at least two elements. Although associations and generalizations fall into the definition of a dependency, they have special names and semantics.

> “A dependency exists between two elements if changes to the definition of one element (the supplier) may cause changes to the other (the client). With classes, dependencies exist for various reasons: One class sends a message to another; one class has another as part of its data; one class mentions another as a parameter to an

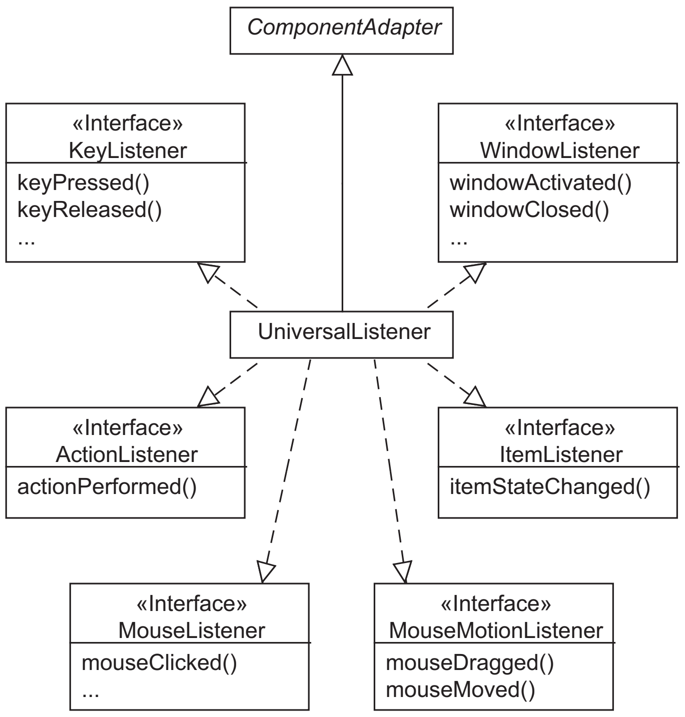


Figure 2.10: A class inheriting from one superclass and implementing multiple interfaces

> operation. If a class changes its interface, any message sent to that class may no longer be valid." [20, page 47]

Dependencies are represented by dashed lines, and a *stereotype* classifying it. If it is directed it features an open arrow head (see figure 2.11).

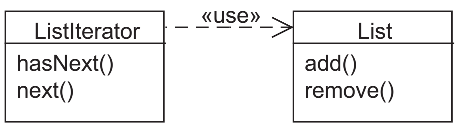


Figure 2.11: Dependency

Constraints: UML is not very restrictive about how to specify constraints. The only rule is to use surrounding brackets. There is no special syntax or guidelines. Fowler[20] suggests using natural language to avoid misinterpretation:

> "You can use natural language, a programming language, or the UML's formal Object Constraint Language (OCL) [...], which is based on predicate calculus. Using a formal notation avoids the risk of misinterpretation due to ambiguous natural language. However, it introduces the risk of misinterpretation due to writers and reader not really understanding OCL. So unless you have readers who are comfortable with predicate calculus, I'd suggest using natural language." [20, pages 49–50]

Figure 2.12 shows constraints on a class and on one of its attributes.

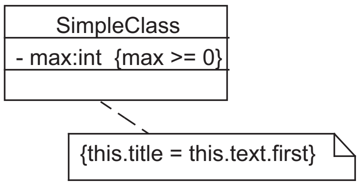


Figure 2.12: Constraints

Templates allow one to specify the type of parameters of an element. Usually templates are used to specify the types of elements of lists, sets and collections.

Languages like C++ and Java since its 1.5 specification have support for templated classes also called *parameterized* classes.

UMLet uses parameterized classes where applicable. The following example presents a `HashMap` that's key and value parameters are bound to the types of `String` and `ElementObject`. See also figure 2.13

```
private HashMap<String,ElementObject> elementsH;
```

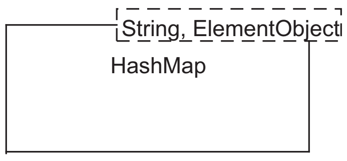


Figure 2.13: Templated class

> "A template is the descriptor for an element with one or more unbound formal parameters. It defines a family of potential elements, each specified by binding the parameters to actual values. Typically, the parameters are classifiers that represent attribute types, but they can also be integers or even operations." [38, page 639]

To express that a class is templated there is a dashed rectangle on the upper right corner of the class element.

Active classes indicate that each instance has its own thread of control, like processes of operating systems that are encapsulated and protect their internals and which have their own memory space. Active classes differ from normal classes by the additional vertical lines in their graphical representation (figure 2.14).

Component diagram

Component diagrams are used to show the components of a system, how they are broken down and how they interact. The components are connected via interfaces using the ball-and-socket notation.

> "The component diagram shows the organizations and dependencies among components and artifacts. The components represent cleanly grouped and encapsulated elements from the logical architecture. The components are typically implemented as files in the development environment; these are modeled as artifacts." [17, page 272]

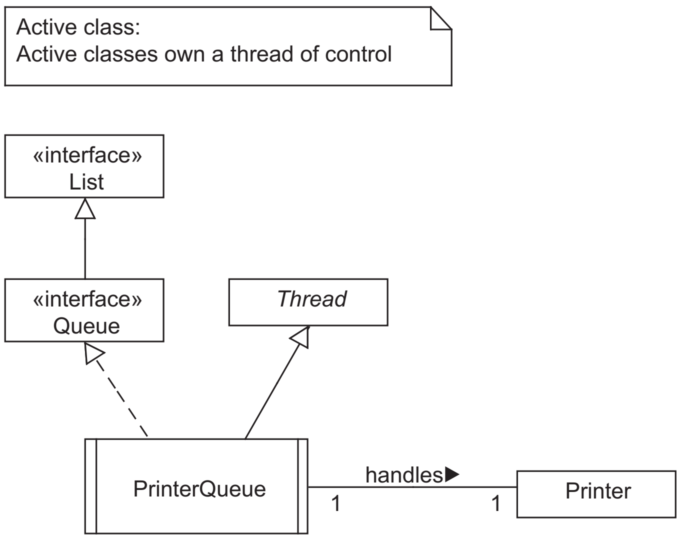


Figure 2.14: Active class

Fowler [20] states that component diagrams are an aid for marketing representatives that help them in communicating and selling components to customers. Because of that the decomposition of the system is not only a technical question but also a question of marketing.

> "The important point is that components represent pieces that are independently purchasable and upgradeable. As a result, dividing a system into components is as much a marketing decision as it is a technical decision [...]." [20, page 141]

Composite structure diagram

The composite structure diagram shows the runtime architecture of a composite element or a system of collaborating objects. It depicts the parts of a structure – how object instances are composed together forming the structure. See figure 2.16 for an example.

> "A diagram that shows the internal structure (including parts and connectors) of a structured classifier or collaboration. There is no rigid line between a composite structure diagram and a general class diagram." [38, page 264]

> "Though similar to a class diagram, a composite structure diagram shows parts and connectors. The parts are not necessarily classifiers in the model, and they do not represent particular instances;

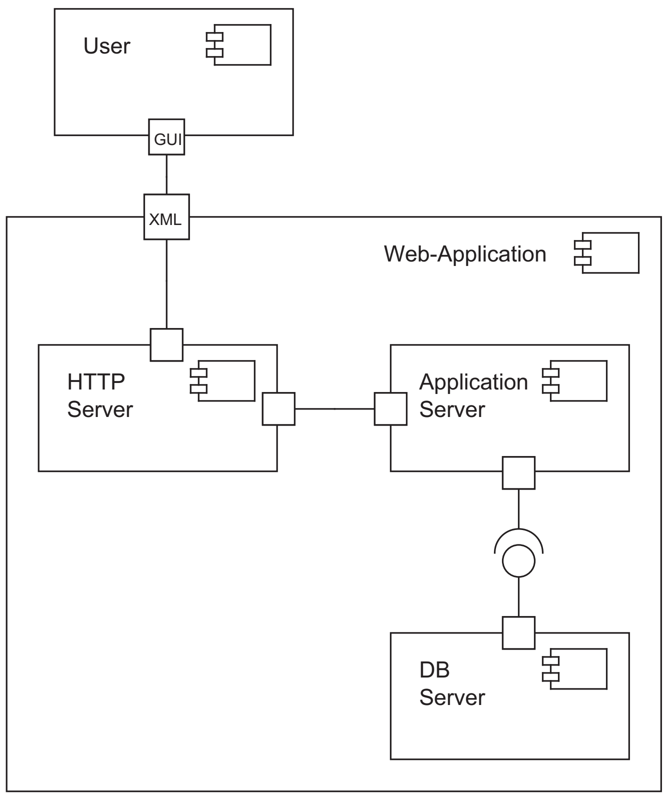


Figure 2.15: Component diagram

they are roles that classifiers will play. Parts are shown in a similar manner to objects, but the name is not underlined. The diagram specifies the structural features that will be required to support the enclosing classifier." [17, page 258]

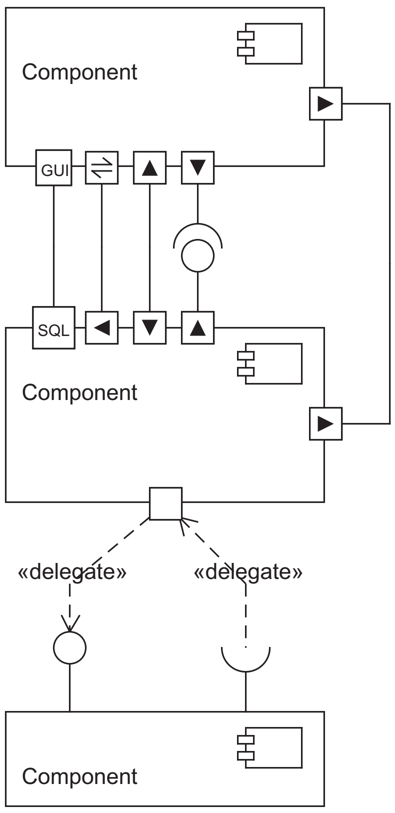


Figure 2.16: Composite structure diagram

Deployment diagram

The deployment diagram is used to show the artifacts and devices of runtime and execution environments.

"Deployment diagrams show a system's physical layout, revealing which pieces of software run on what pieces of hardware." [20, page 96]

> "It is the ultimate physical description of the system topology, describing the structure of the hardware units and the software that executes on each unit. In such an architecture, it should be possible to look at a specific node in the topology, see which components are executing in that node and which logical elements (classes, objects, collaborations, and so on) are implemented in the component, and finally trace those elements to the initial requirement analysis of the system (which could have been done through use-case analysis)." [17, page 273]

The main actors of this diagram type are nodes and communication paths. Nodes represent real (*device*) or virtual hosts (*execution environment*) like computers, mobile devices or virtual machines and interpreters capable of executing software. Inside nodes artifacts are physical representations of software like class files, JARs, executables or scripts and also databases meaning that this artifact is deployed on that node at runtime.

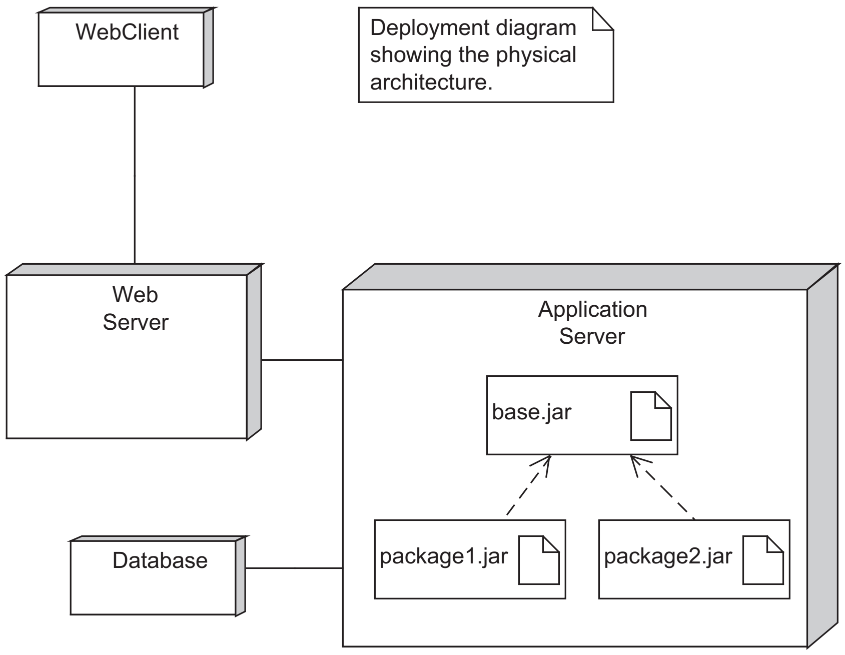


Figure 2.17: Deployment diagram

Object diagram

Object diagrams are similar to class diagrams, but they focus on instances of classes and their dynamic characteristics using mostly the same notation.

"An object diagram is a snapshot of the objects in a system at a point in time. Because it shows instances rather than classes, an object diagram is often called an instance diagram." [20, page 87]

Object diagrams do not define a system, but are a method of giving examples, assisting the understanding of a system's definition.

"Snapshots are examples of systems, not definitions of systems. The definition of system structure and behavior is the goal of modeling and design. Examples can help to clarify meanings to humans, but they are not definitions." [38, page 67]

To express the difference to the static class diagram, object names are underlined and may contain detailed value specifications.

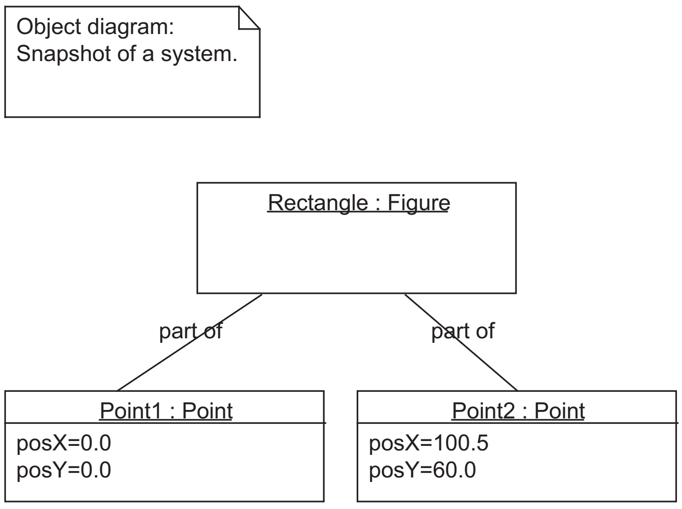


Figure 2.18: Object diagram

Package diagram

Packages group together UML elements into a superior structure for managing larger systems. Package diagrams are not bound to classes but work together with various UML elements, although they are mainly used with classes.

Package diagrams have a hierarchic structure meaning that packages can contain nested packages. Packages provide a namespace to their elements. One element may be contained in one package only but can be uniquely referenced using a qualified name including the sequence of all names of the packages above.

The contents of packages may be marked with visibility modifiers as it is done in the class diagram.

The package diagram is useful to show the dependencies between the elements of large systems. Package diagrams directly correspond to Java's package scheme or to the namespace concept of C++ and .NET. The UML defines rules how package diagrams are to be used. But Fowler [20] states that users should bend these rules to match the actual needs reducing the risk of misunderstandings.

> "Different programming environments have different rules about visibility between their packaging constructs; you should follow the convention of your programming environment, even if it means bending the UML's rules." [20, page 90]

UML 2 offers two notations of the package diagram.

A package is drawn as a rectangle with a small rectangle attached to it like the common folder icon. One way to show the contents of a package is to place the elements right into the package. The other option is to attach the proposed elements by using a branched line with a circle having a small cross (or plus). See figure 2.19 for examples.

UMLet assists the second way of defining a package diagram with a special element using grammar. The grammar to the package hierarchy on the image is:

```
java.util
concurrent
>atomic
locks
<jar
logging
prefs
regex
zip
```

Activity diagram

The activity diagram is related to the state machine diagram (also: state chart diagram) and is capable of modeling parallel behavior and is concerned with activities, their inputs and outputs. It shows the flow of activities and defines the sequence of execution.

> "Activity diagrams focus on the dynamic flow of a system. While interaction diagrams focus on messages between instances and state machines show the life-cycle progress of a classifier, activity diagrams capture movement and process without having to reference a class or object." [17, page 207]

> "Activity diagrams are a technique to describe procedural logic, business process, and work flow. In many ways, they play a role similar to flowcharts, but the principal difference between them and flowchart notation is that they support parallel behavior." [20, page 117]

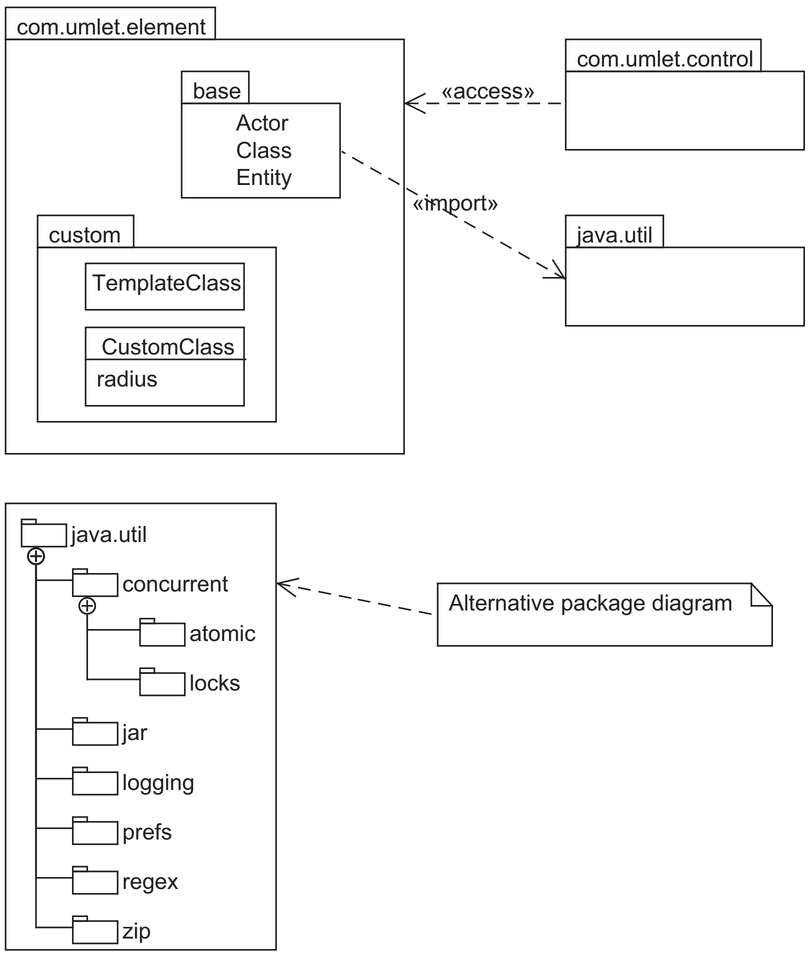


Figure 2.19: Package diagrams

The basic elements of an activity diagram are actions and edges describing the flow of control. An activity refers to a sequence of actions meaning that the activity diagram shows a single activity composed of a set of sequential actions.

Actions are shown as rectangles with rounded edges containing a naming string. The edges which represent transitions are drawn as arrows with optional information and constraints like a condition in combination with a decision branch. Additional elements are heavy bars depicting forks and joins, decisions and merges represented as diamonds and elements for sending and receiving signals. The beginning and ending of an activity diagram are normally depicted with an initial node shown as a filled circle and the final node depicted as a circle surrounding a filled circle.

Within systems utilizing parallelism transactions have to be synchronized which is modeled with heavy bars representing AND-conditions while merges, drawn by diamonds or just by simultaneous incoming edges, represent OR-conditions. Another special element is the *flow end* to show the end of a flow in concurrent actions, while the other flows or the superior activity is not affected. The flow end is drawn as a circle having a cross within.

UMLet provides special assistance to the creation of activity diagrams. With a simple but powerful syntax activity diagrams may be textually built. The diagram in figure 2.20 is derived by the following syntax:

```
title:activity diagram
start->if(value == true){action1:a1}(else){action2:a2}
a1->join:j1
a2->j1
j1->forkH:f1{action3:a3}{action4:a4}{action5:a5}
a3->fend:fe
a4->syncH:s1
a5->s1
s1->end
```

Use case diagram

> "Use cases are a technique for capturing the functional requirements of a system. Use cases work by describing thy typical interactions between the users of a system and the system itself, providing a narrative of how a system is used." [20, page 99]

Use cases are sets of scenarios to solve a common goal. Users are referred as actors though they don't necessarily have to be human since an external computer system may act as an actor.

Use cases help to describe the functional requirements of a system represented as an external view onto the system. This implies that a use case does not need to have correlations to classes or other fine grained system internals.

Use case diagrams consist of actors, use cases and a system, but the UML makes no hard constraints on the set of allowed elements since the goal of modeling is to create simple to understand but detailed models.

Actors are usually represented by class like entities with the stereotype <<*actor*>> or by *stickmen*. Use cases are drawn with an ellipsis containing the name of the use

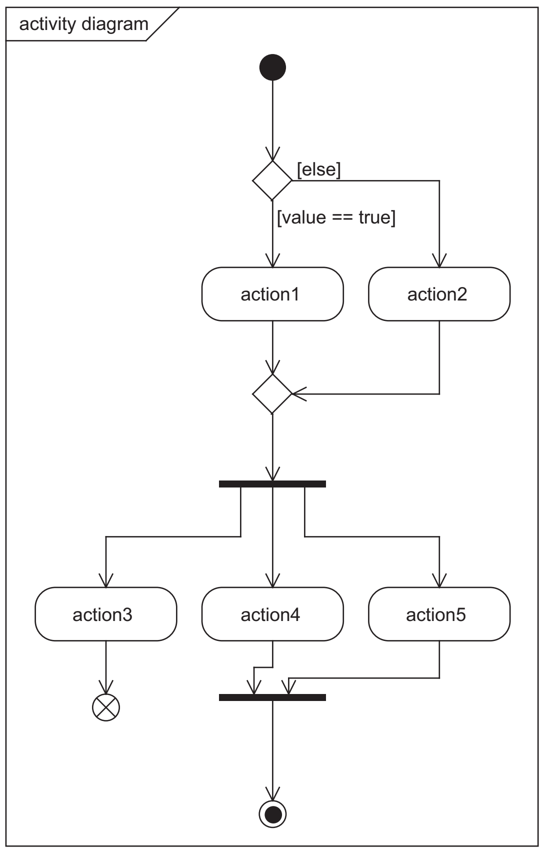


Figure 2.20: Activity diagram

case represented. Optionally the name of the use case may be placed below the ellipsis. The system is drawn as a rectangle containing the use cases while the actors interact with the system by participating in one or more use cases.

The relationships between use cases include

- associations
- extend relationships
- include relationships
- generalizations

The include relationship means that the included use case is part of another use case. Extend relationships describe an incremental extension of a base use case meaning that the extended use case is defined independently of the extending use case. So while the included relationship describes a mandatory inclusion behavior, the extends relationship is an optional inclusion — the extended use case may, but does not have to make use of the extension use case.

State machine diagram

> “All objects have a state; the state is a result of previous activities performed by the object and is typically determined by the values of its attributes and links to other objects. A class can have a specific attribute that specifies the state, or the state can be determined by the values of the ’normal’ attributes in the object.” [17, page 247]

The state machine diagram (also referred as *state chart diagram*) describes a state machine with it’s simple states, state transitions and composite states. State machines may describe the behavior of objects using states and state transitions triggered by external or internal events. The actions and activities of the object are fired by the transitions.

> “State machines may be used to describe user interfaces, device controllers, and other reactive subsystems. They may also be used to describe passive objects that go through several qualitatively distinct phases during their lifetime, each of which has its own special behavior.” [38, page 35]

The state machine diagram utilizes states (rectangles with rounded borders), forks and synchronization (heavy bars), decisions (diamonds) and start and end states (filled circles and filled circles surrounded by a circle).

State changes may have guarding conditions that have to be true in order to fire the transition. If a state offers more than one state transition, guarding conditions which have to be mutually exclusive are used to select one single transition if applicable. If within a state an event occurs that is not applicable to any of the transitions leading away from the state, nothing happens and the state is retained.

The activities of the object (like methods) are activated and performed between the states while the state transition is in progress - in contrast to the activity diagram, where the activities happen within the activities. These activities do not have to be simple in any way, but can implicate time and resource consuming procedures.

The state machine diagram starts at that state marked with the start state also called initial pseudo state.

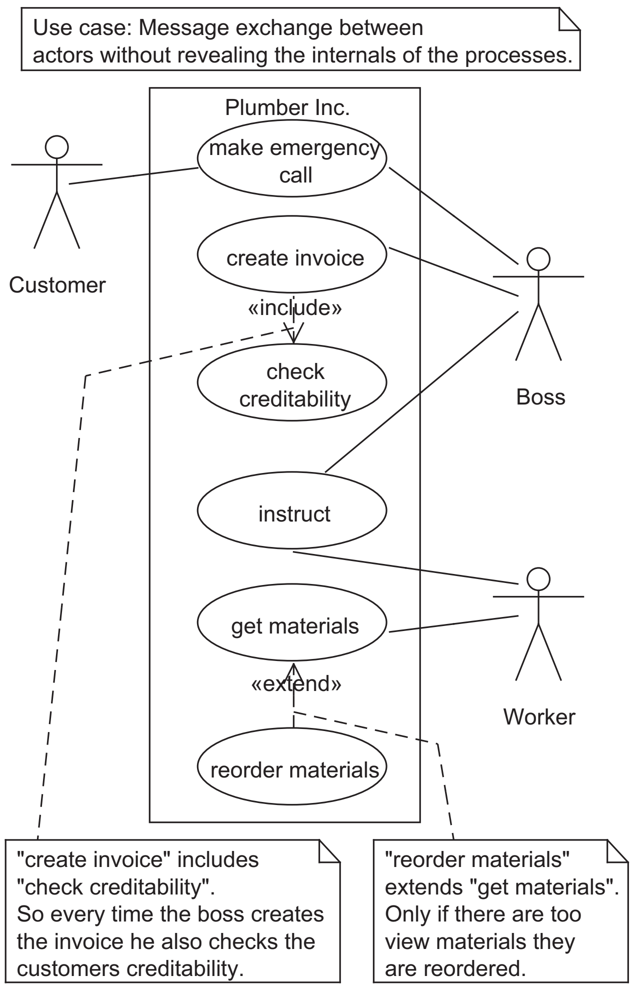

Use case: Message exchange between actors without revealing the internals of the processes.
Plumber Inc.
make emergency call
create invoice
«include»
check creditability
instruct
get materials
«extend»
reorder materials
Customer
Boss
Worker
"create invoice" includes "check creditability". So every time the boss creates the invoice he also checks the customers creditability.
"reorder materials" extends "get materials". Only if there are too view materials they are reordered.

Figure 2.21: Use case diagram

States can have *internal activities* and *internal transitions*: Internal activities describes internal behavior triggered by events. There are three reserved standard events defined: *entry* for activities to be done when entering the state, *exit* defines the activities done before leaving the state and *do* defines activities that are performed while residing in the state like searching, polling or calculating. Internal transitions initiate activities that do not involve a state change, so they never trigger *entry* or *exit* activities.

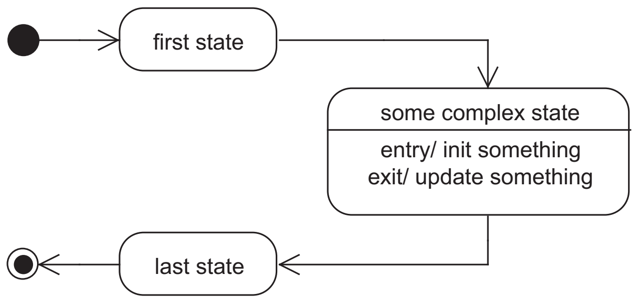


Figure 2.22: State machine diagram

Sequence diagram

Sequence diagrams are the most common kind of interaction diagrams focusing on the sequence of interactions between affected objects and show the life time of objects and their communication using messages within a use case. They allow a very detailed description of how the objects interact with each other.

Every object is defined by a box with the object's name, a lifeline heading from the top to the bottom of the diagram and boxes showing its activity. The messages are drawn with horizontal arrows pointing from the sender to the receiver. If the receiver of the message is equal to the sender the arrow may be drawn as an arc or a rectangular shape beginning and ending in the same object. The appearance of messages defines their semantics: While asynchronous messages are drawn with an open arrow head, synchronous messages have a filled arrow head and the method response messages feature a dashed line. Interaction frames allow the grouping of interactions allowing them to be looped or requiring entry conditions.

> "One use of a sequence diagram is to show the behavior sequence of a use case. When behavior is implemented, each message on a sequence diagram corresponds to an operation on a class or an event trigger on a transition in a state machine." [38, page 38]

Like with the activity diagram UMLet provides special support to this type of diagram by a textual description language.

```
title:sequence diagram
_alpha:A_|_beta:B_|_gamma:G_
```

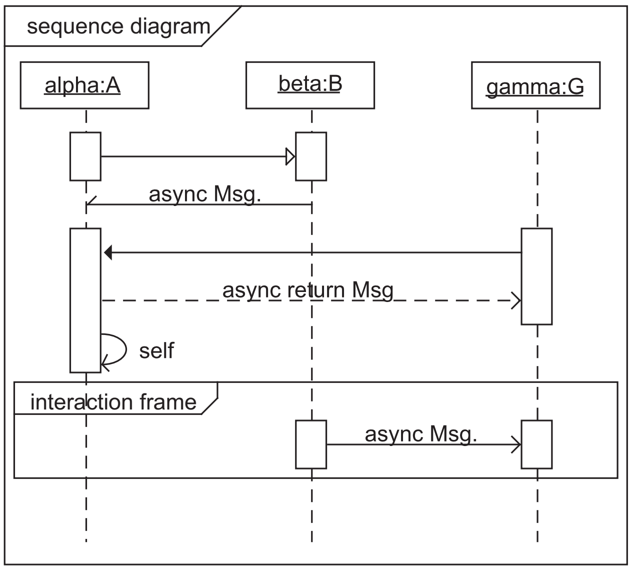


Figure 2.23: Sequence diagram

```
1->>2:1,2
2-/>1:async Msg.
3->>>1:1,3
1.>3:1,3:async return Msg
1->1:1:self
iframe{:interaction frame
2->3:2,3:async Msg.
iframe}
```

Communication diagram

The communication diagram is related to the sequence diagram since it focuses on how objects interact with each other and in what order they send and receive messages. The obvious difference between the two diagrams is that the communication diagram has no time line and no limitation on the arrangement of the objects so they can be placed the way they fit best and may show best their relation to other objects.

Messages are modeled as lines between the affected objects without arrow heads. Instead a small arrow is placed aside the line depicting the direction of the message. To define the sequence the messages are sent, each message gets a number.

> "Communication diagrams can be used to show quite complex interactions between objects. However, learning the numbering scheme

of messages can take some time, but once learned, it is rather easy to use. The main difference between communication diagrams and sequence diagrams is that communication diagrams show the actual objects and their relations (the "network of objects" that are communicating with their relations implied by the paths of communication), which in many situations can ease the understanding of the interaction." [17, page 185]

The syntax of the message labels is quite complicated and follows this scheme (see [17, pages 183–184] for further details):

```
[predecessor] [guard-condition] sequence-expression return-value

predecessor := sequence-number [',' sequence-number]*
guard-condition := '[' condition-clause ']'
sequence-expression := [integer | name] [recurrence]
recurrence := ( '*' '['iteration-clause']' ) | ( '['condition-clause']' )
```

The predecessor is used for synchronization, meaning that all messages listed with their sequence number have to be performed first. The guard-condition is used to express that the message is only sent, if the stated condition is true. The sequence-expression shows in what order the messages are performed. Additional iteration may be modeled like `*[i=1..n]`and the condition option may be used to model branches. The return value should be assigned to a message signature (method signature) like `a=calc(b, c)`. Figure 2.24 shows a communication diagram with sequences and a message having a guard-condition.

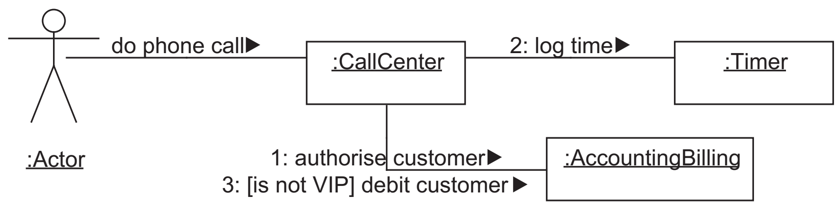


Figure 2.24: Communication diagram

Communication diagrams were called collaboration diagram before UML 2.0.

Interaction overview diagram

The interaction overview diagram is a specialized variant of the activity diagram representing interactions. Like the name of this diagram type indicates, it provides an overview of the flow of control in a system. The nodes of the diagrams are interactions or interaction uses. While interactions can be any type of interaction diagrams like an activity diagram, a sequence diagram or a communication diagram, interaction uses are named references to interaction diagrams.

Interaction overview diagrams are a mixture of activity diagrams and sequence diagrams.

> "This is a strange mixture of concepts from activity diagrams and sequence diagrams. It tries to mix the control flow mechanism among nodes from activity diagrams with the sequence of messages among lifelines from sequence diagrams." [38, page 412]

Timing diagram

The timing diagram is used to show object interactions and state changes along the linear time axis. It describes the state changes over time stimulated by external or internal events of the observed objects.

> "A timing diagram provides a convenient way to show active objects and their state changes during their interactions with other active objects and system resources. The X-axis of the timing diagram has the time units, while the Y-axis shows the objects and their states." [17, page 230]

[38, page 653] say that timing diagrams are an alternative way of displaying sequence diagrams that shows state changes explicitly on a lifeline. The differences to the standard sequence diagram are:

- The diagram is drawn horizontally from left to right. The time axis corresponds to the x-axis.
- The lifelines are arranged vertically in separate compartments.
- Lifelines are not straight linear, but perform level jumps to show state changes. Every level represents a unique state.
- Alternatively lifelines may follow a straight line with different state or value marks.
- Tick marks may define time intervals and/or discrete events on an optional metric time axis.
- The times of the different lifelines are synchronized.
- Values of objects may be given.

The timing diagram is widely used to describe complex processes in real-time systems, and digital electronics to show state changes with optional time constraints like *slew rates*. This is the maximum rise time logical gates need to change their logical state from *low* to *high*.

UMLet has no support for this type of diagram yet, since its way of composition is different to all other diagrams.

2.1.3 Views

Another way of approaching the UML and it's diagrams are the so called views. Although they are not part of the UML standard itself they help to use the UML and make the challenge of describing a system easier. Views provide a toolkit for describing the system from a specific point of view. Since the stakeholders like customers, system analysts and developers concentrate on different aspects, the proposed views help to cover the respective aspects.

> "A system description requires a number of views, where each view represents a projection of the complete system that shows a particular aspect." [17, page 21]

Each view suggests the use of a set of diagram types for description, but these diagram types are not mutually exclusive, so one diagram type may be used in more than one view.

> "By looking at the system from different views, it is possible to concentrate on one aspect of the system at a time. A diagram in a particular view needs to be simple enough to communicate information clearly, yet coherent with the other diagrams and views so that the complete picture of the system is described by all the views put together." [17, page 21]

Since the view concept is not part of the UML itself different authors and tools offer different or additional views, than the ones presented here, concentrating on further points of view to a system.

1. Use-Case view: The use-case view provides tools and methods for describing and documenting a system by investigating the system from the viewpoints of system external actors. Use cases allow the actors of the system to be human users or other computer systems. While in the requirements analysis phase the use-case view is utilized by system analysts and customers it is also used in the testing phase by testers to verify the systems with the stated requirements.

   > "The use-case view is central, because its contents drive the development of the other views. The final goal of the system is to provide the functionality described in this view —- along with some nonfunctional properties. Hence, this view affects all the others. This view is also used to validate the system and finally to verify the functioning of the system by testing the use-case view with the customers (asking, "Is this what you want?") and against the finished system (asking, "Does the system work as specified?")." [17, page 23]

2. Logical view: The basic structure of a system and its functionality is aggregated by the logical view and diagrams. The designers and developers of a system use the methods of this view to describe and design the static parts of the system and their collaborations.

   > "In contrast to the use-case view, the logical view looks inside the system. It describes both the static structure (classes, objects, and relationships) and the dynamic collaborations that occur when the objects send messages to each other to provide a given function. Properties such as persistence and concurrency are also defined, as well as the interfaces and the internal structure of classes. The static

> structure is described in class and object diagrams. The dynamic modeling is described in state machines, and interaction and activity diagrams." [17, page 23]

3. Implementation view: The implementation view is not very fine grained. It concentrates on the main modules of the system and their interactions.

   > "It is mainly for developers and consists of the main software artifacts. The artifacts include different types of code modules shown with their structure and dependencies. Additional information about the components, such as resource allocation (responsibility for a component) or other administrative information, such as a progress report for the development work, can also be added." [17, page 23]

4. Process view: This view allows one to describe how a system is composed from the parts that represent resources or control tasks and processes of the system. It shows how to organize these parts and control and synchronize them to work together.

   > "The emphasis on a view that shows concurrency provides critical information for developers and integrators of the system. The view consists of dynamic diagrams (state machines, and interaction and activity diagrams) and implementation diagrams (interaction and deployment diagrams)." [17, page 23]

5. Deployment view: The deployment view shows how the system is fragmented and distributed to the execution environments and the communication among them.

   > "The deployment view is used by developers, integrators, and testers and is represented by the deployment diagram. This view also includes a mapping that shows how the artifacts are deployed in the physical architecture, for example, which programs or objects execute on each respective computer." [17, page 24]

2.1.4 Summary

The UML is a standardized modeling language controlled by the OMG that has developed from different methods of describing systems and requirements. After the software industry recognized UML's potential the UML Partners consortium was joined by a number of well known companies representing broad acceptance.

The UML provides a set of tools to describe a system and it's details. The UML 2.0 standard features 13 different diagram types specialized to cover the respective structural and behavioral aspects of the system.

UML is not like strictly constrained law, but it is an agreed standard that provides notations allowing the users to adapt in a way that may ease the understanding of what is meant.

Views like those proposed by [17] and others help to approach the task of describing a system by observing it from different points of view, since different participants have different angles to the system.

2.2 Design patterns with UMLet

This chapter will present design patterns and why they are used within UMLet: The fact that the structure of design patterns may be easily illustrated using UMLet is not the primary idea of this chapter. The more interesting question is how the knowledge of design patterns affects the design and implementation of UMLet and what users of UMLet gain from a structured and clear design.

The question of why UMLet is developed with special respect to design patterns is as follows: On one hand using design patterns eases the development. And on the other hand design patterns help to explain the structure of UMLet and ease the understanding of the source codes since UMLet is released as open source. So users may freely adapt, modify and extend UMLet to their needs.

One goal of design patterns is structuring the design of software components. A software system gets very hard do maintain if the objects are too closely engaged. Good software design and design patterns help to decouple the involved objects and provide strong structures and procedures.

> "In other words, design patterns describe how objects communicate without become entangled in each other's data models and methods. Keeping this separation has always been an objective of good OO programming, and if you have been trying to keep objects minding their own business, you are probably using some of the common design patterns already." [12, page 11]
>
> "Design patterns help to improve communication software quality since they address a fundamental challenge in large-scale software development: communication of architectural knowledge among developers." [39]

It is not enough to write code in a common and widely accepted programming language. The use of a structured design helps to get into an existing software system. Design patterns provide the basis for structuring the components of the software system, provide standardized terms and allow thinking and discussing on a higher level.

> "A design pattern represents a recurring solution to a design problem within a particular domain (such as business data processing, telecommunications, graphical user interfaces, databases, and distributed communication software) [1]. Design patterns facilitate architectural level reuse by providing "blueprints" that guide the definition, composition, and evaluation of key components in a software system. In general, a large amount of experience reuse is possible at the architectural level. However, reusing design patterns does not necessarily result in direct reuse of algorithms, detailed designs, interfaces, or implementations." [39]

Often the book "Design Patterns: Elements of Reusable Object-Oriented Software" by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [22] (also known as the *Gang of Four*, or *GoF*) is referred as the birth of mainstream design patterns. But design patterns have been used before that time: Christopher Alexander described patterns applied to the construction of towns and buildings in the 1970s.

> “Cristopher Alexander says, ’Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same way twice’ [AIS+77, page x]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls and doors, but at the core of both kinds of patterns is a solution to a problem in a context.” [22, pages 2–3]

> “I like to think of a pattern as a best practice solution to a common recurring problem. That is, a pattern documents and explains an important or challenging problem that can occur when designing or implementing an application, and then discusses a best practice solution to that problem. Over time, patterns begin to embody the collective knowledge and experiences of the industry that spawned it.” [31, page XIV]

Since the publication by Gamma et. al.[22], design patterns have been a common discussion topic of software developers around the world. ALur et. al. [1] state that design patterns are not developed or invented by a specific person or organization, but design patterns are discovered within recurring software designs. By identifying, naming and documenting the patterns these authors provide a high value to the software developer community.

> “Of course, while the Gang of Four work resulted in patterns becoming a common discussion topic in software development teams around the world, the important point to remember is that the patterns they describe were not invented by these authors. Instead, having recognized recurring designs in numerous projects, the authors identified and documented this collection.” [1, page 9]

Because Gamma et. al.[22] was pretty much the first work on design patterns in the field of software engineering, these design patterns are well known to most software developers. But there are also more authors and books with interesting and important design patterns.

It is worth mentioning that design patterns are not referred to by a single name only, as a result of the development process of software engineering itself. Although it is recommendable to agree on a common name, there are several names describing the same pattern: The *abstract factory* is also known as *kit* while the *factory method* sometimes is called *virtual constructor*. The *decorator* pattern may be found as *wrapper* and the *chain of responsibility* is referred to as *event handler*, *bureaucrat* and *responder*.

Gamma et. al. [22, page 3] state that a design pattern is basically defined by four elements:

1. Name: The name of a design pattern usually describes the particular design problem, its solution and consequences in short. Naming allows designing and discussing at a higher level of abstraction.
2. Problem: The problem defines the problem space, the implications, the conditions and the context to which the design pattern applies to.
3. Solution: Describes the way the design pattern solves the problem, which elements are used, their relations, responsibilities and collaborations. A design

pattern is like a template and provides an abstract description of a design problem and how to solve it.

4. Consequences: Designing patterns describing general problems imply trade-offs when applying to specific problems. These trade-offs, costs and benefits are described by the consequences allowing to evaluate the impact on the system.

> "A design pattern names, abstracts, and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and interfaces, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use." [22, pages 3–4]

Following Gamma et. al. [22] design patterns can be divided into the three major groups:

1. *Creational patterns* help designing for the process of instantiating classes.

   > "Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object." [22, page 81]

2. *Structural patterns* combine classes and objects into larger structures. While structural class patterns deal with classes at compile-time, describing interfaces and implementations by inheritance, structural object patterns combine objects to form new functionality and may be modified at run-time.

   > "Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. [...] Rather than composing interfaces or implementations, structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition." [22, page 137]

3. *Behavioral patterns* do not only describe class and object relations but they describe the communication and responsibilities among these classes and objects.

   > "Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected." [22, page 221]

2.2.1 Design patterns used within UMLet

As stated at the beginning of this chapter UMLet was designed to be modified and extended by its users. In order to ease these intentions UMLet's design features a set of well known design patterns. The following section will illustrate the used design patterns.

Prototype

A prototype is used when the instantiation of a class is very expensive (time consuming or resource intensive) or the system has no knowledge of how objects are created — applications that load classes dynamically cannot refer to the classes constructor statically. The object creation is replaced by object cloning and modification. But prototyping is sometimes hard or even impossible to implement. Adding a cloning method may be impossible with classes that already exist. Classes that have internals that don't support cloning or implement circular references can not be cloned. The prototype pattern belongs to the creational patterns.

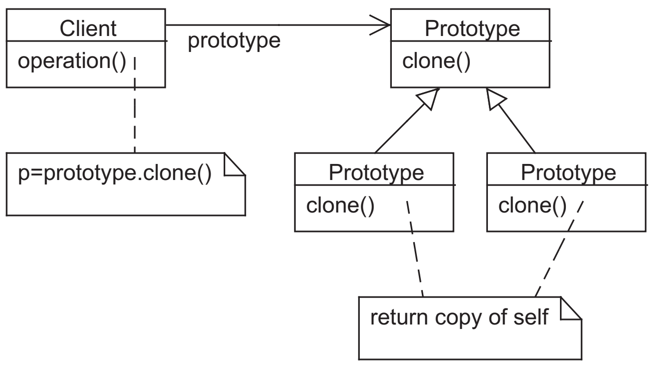


Figure 2.25: Prototype

UMLet uses the prototype pattern to create new instances of its entities (graphical elements). The application framework simply asks the entity to create a new instance of itself using the `cloneFromMe()` method which is defined in the superclass:

```
public Entity CloneFromMe() {
  try {
   //get class of dynamic object
    java.lang.Class cx= this.getClass();
    Entity c = (Entity)cx.newInstance();
    c.setState(this.getPanelAttributes()); //copy states
    c.setBounds(this.getBounds());
    c.sourceFilePath=this.sourceFilePath;
    c._javaSource = this._javaSource;
    return c;
  } catch (InstantiationException e) {
    System.err.println("UMLet->Entity/"+
                     this.getClass().toString()+": "+e);
  } catch (IllegalAccessException e) {
    System.err.println("UMLet->Entity/"+
                     this.getClass().toString()+": "+e);
  }
  return null;
}
```

Note the use of Java's reflection to gain the type of the actual object to be cloned. One could expect that `Entity` just provides an abstract `cloneFromMe()` method to be overridden by it's subclasses. This is just for convenience, allowing one to spare this method in the custom element template used for extending UMLet.

Singleton

This creational pattern is one of the most commonly used patterns. It is used to provide and ensure a single instance of a class.

UMLet makes intensive use of the singleton design pattern, to express that there is only one instance of an object. Like UMLet's main class `Umlet`, the event listener implementation `UniversalListener` and the command processor `Controller` are singletons.

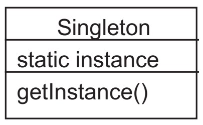


Figure 2.26: Singleton

```
private UniversalListener() {} //hide the constructor

private static UniversalListener _instance;
public static UniversalListener getInstance() {
  if (_instance==null) _instance=new UniversalListener();
  return _instance;
}
```

When considering the use of the singleton pattern the question arises if it is better to uses eager initialization or lazy initialization. While eager initialization requires the extra resources from the beginning, with lazy initialization the resources may not be occupied if the singleton is never requested. Another point is the time needed for initialization: If it is a time-consuming process to create the object, eager initialization may be the choice within time critical software systems. When the singleton is implemented using lazy initialization special care has to be taken in multi threaded applications. There has to be a concurrency control to avoid multiple initialization of the singleton breaking the singleton pattern concept. Concurrency control may be implemented in Java by synchronizing the method.

Adapter

The adapter pattern is a structural pattern. Adapters provide a class with an additional interface enabling it to work together with classes with incompatible interfaces, by using inheritance or object composition.

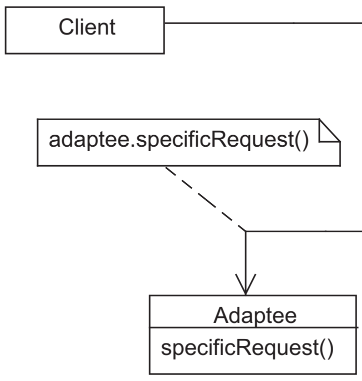


Figure 2.27: Adapter

UMLet implements this design pattern to be used as an Eclipse-IDE plug-in. The `UmletEditor` class integrates UMLet as an Eclipse editor into the development environment. UMLet provides its functionality on one side, while the Eclipse framework offers a standardized interface. To make both interfaces work together the adapter pattern is the first choice.

Composite

The composite pattern is a structural pattern and offers to handle a composition of objects in the same way as a single entity of such an object. It represents the primitive object and its container in an abstract class.
Like with trees there are nodes that have children and nodes that don't. The composite pattern helps to simplify implementation since there is no need to distinguish between compositions and simple types.

UMLet uses this pattern with the `Container` class of the `java.swing` package to combine sets of entities. The container is handled by the `paint()` method of the drawing panel like a single entity.

Command

The command pattern belongs to the behavioral patterns. A command encapsulates requests to an object into a command-object offering flexible execution of the requests. These command objects may be stored into FIFO (first in first out) structures and used to implement a do-undo functionality.

The do-undo functionality of UMLet is realized using the command pattern. Simple commands are wrapped into command-objects and processed by the command

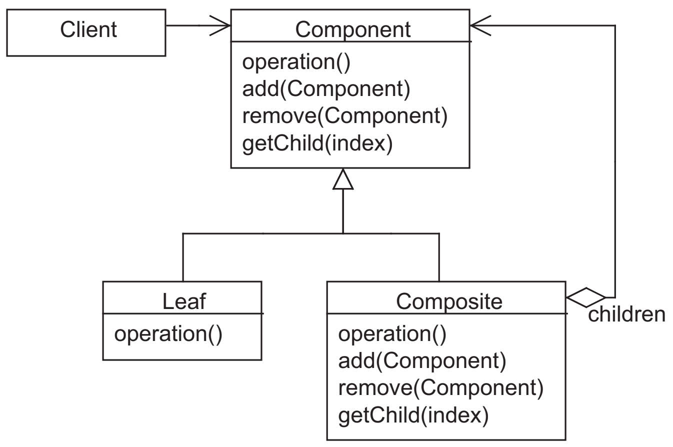


Figure 2.28: Composite

processor.

```
public abstract class Command {
  public void execute() {
    Umlet.getInstance().setChanged(true); //set diagram changed flag
  }
  public void undo() {
    Umlet.getInstance().setChanged(true);
  }
}

public class Controller {
  private Vector<Command> commands=new Vector<Command>(); //command queue

  public void executeCommandWithoutUndo(Command newCommand) {
    newCommand.execute();
  }
  public void executeCommand(Command newCommand) {
    [...]
    //add the command to the command queue for possible undo
    commands.add(newCommand);
    newCommand.execute();
    [...]
  }
}
```

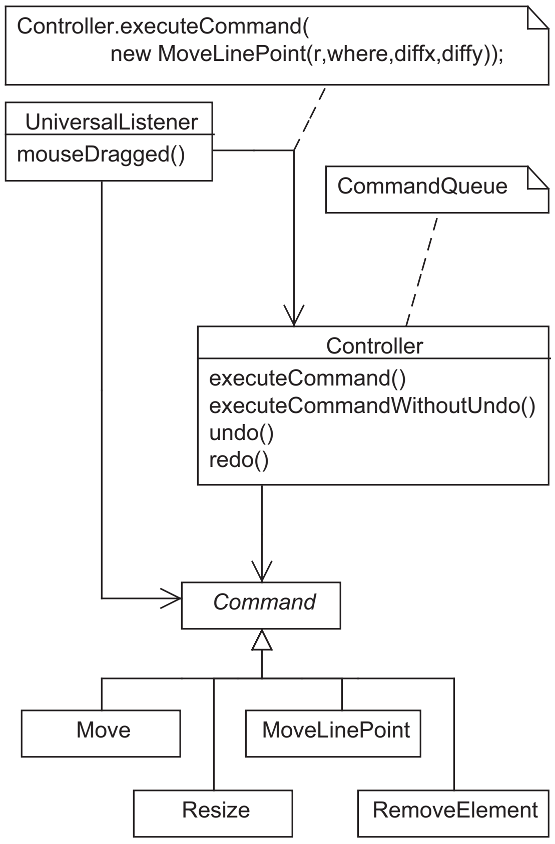


Figure 2.29: Command

Iterator

An iterator offers an alternative to iterate through a set of objects without knowing the implementation of the underlying aggregation object. There may be also specialized iterators that interpret or filter the contents returning only specific elements of the set. This pattern belongs to the group of behavioral patterns.

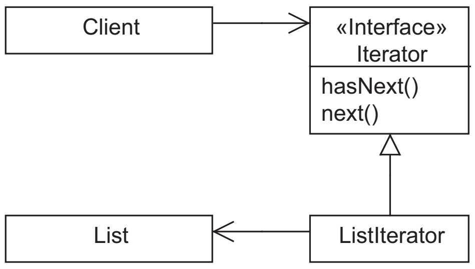


Figure 2.30: Iterator

Java offers the `Iterator` interface to provide defined methods for iterating over object sets like lists, sets and queues. As an example UMLet uses the iterator to iterate over the set of interactions within the sequence diagram using the `hasNext()` and `next()` methods provided by the iterator.

```
private Set[]<Interaction> level; //the set of interactions
public boolean controlBoxExists(int levelNum, int objNum) {
  Iterator<Interaction> it = level[levelNum-1].iterator();
  while (it.hasNext()) {
    Interaction ia = it.next();
    if (ia.hasControl(objNum)) return true;
  }
 return false;
}
```

Observer

This design pattern enables triggering a state change of a set of related objects by a state change of a single object defining a one–to–many dependency. The observer pattern is commonly used to implement the command-action behavior of modern user interfaces where entities like menu items, buttons, and other gadgets respond to user inputs or data source updates. It is a behavioral pattern.

The observer pattern is utilized by Java's EventListeners. UMLet's `UniversalListener` class implements all listeners for the occurring user interface events like mouse clicks on buttons, menu items and graphical elements and diverse key pressing events. The following code sample illustrates the coupling of the EventListener and the object that fires the events:

```
//create the menu item
```

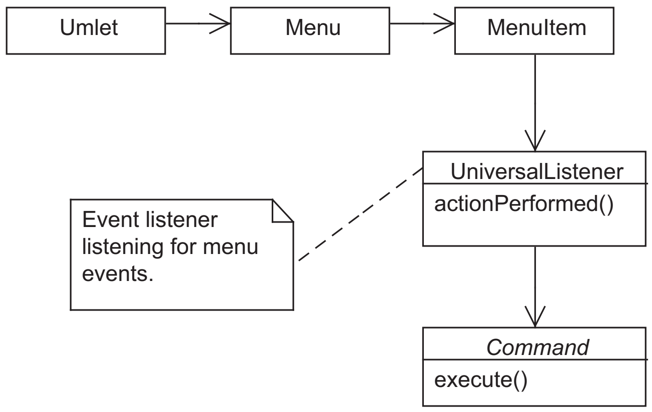


Figure 2.31: Observer

```
JMenuItem iSaveAsPdf=new JMenuItem("Save as PDF..");
//add it to the file menu
iFile.add(iSaveAsPdf);
//connect item with the event listener
iSaveAsPdf.addActionListener(UniversalListener.getInstance());
```

By selecting the ”Save as PDF...” menu item the `actionPerformed()` method of the listener is invoked. It has to distinguish between the possible event sources and delegate the appropriate action.

```
public void actionPerformed(ActionEvent e) {
  //was the event fired by a menu item?
  if(e.getSource() instanceof JMenuItem) {
    JMenuItem item=(JMenuItem)e.getSource();
    if (item.getText().equals("Save as PDF..")) {
      Umlet.getInstance().doSaveAsPdf(null);
    }
  }
}
```

Interpreter

The interpreter describes a definition of a grammar for languages following regular expressions. This pattern is a behavioral pattern.
The interpreter pattern is not a typical design pattern providing a simple explanation of how to interpret and implement it. It rather documents the fact of what is intended to be implemented: a parser for a grammar.

“The Interpreter pattern generally describes defining a grammar for that language and using that grammar to interpret statements in that language." [12, page 145]

"Whenever you introduce an interpreter into a program, you need to provide a simple way for the program user to enter commands in that language. It can be as simple as the Macro record button we noted earlier, or it can be an editable text field [...]." [12, page 153]

UMLet uses this pattern extensively to provide grammars for its elements and diagrams. See the sequence diagram in section 2.1 for a grammar example.

Another example of an interpreter pattern is Java's `java.util.Formatter`. This class is intended to format strings as it is done with the `printf()` function in C:

```
formatter.format("%4$2s %3$2s %2$2s %1$2s",
                 "a", "b", "c", "d") //" d  c  b  a"
```

2.2.2 More design patterns

The following section will present additional design patterns that were not explicitly used when implementing UMLet though some occur within the implementation of Java itself.

Abstract factory

The abstract factory provides an interface for the production of instances of a family of related classes. It is one level above the factory pattern. It is a creational pattern.

An example of an abstract factory is the `Toolkit` class in the `java.awt` package which provides the interface for creating user interface objects like buttons and dialogs. The abstract factory does not provide concrete methods but just defines the interface for the concrete factory whose implementation creates the actual objects.

Factory method

In contrast to the abstract factory the pattern of the factory method provides an interface for the creation of a single product instead of a family of products. It provides an interface for object creation but passes the responsibility of which type is to be created to its subclasses.

Factory methods are often used to implement the *abstract factory pattern*, by providing the concrete factory with the accurate methods for object creation. The factory method pattern is a creational pattern.

Builder

A Builder is a creational pattern and separates the construction of a complex object from its representation. The construction mechanism can build different representations. A builder constructs a composite object by combining objects by the requested scheme, while hiding the composition mechanism.

Bridge

This structural pattern allows the detachment of an implementation from its abstraction. It adds a degree of freedom to the rather inflexible concept of inheritance by letting the implementation develop separately.

Decorator

Just as the adapter pattern adds interfaces to classes, the decorator pattern, which is also a structural pattern, adds additional functionalities and provides a more flexible way of extension than inheritance does.

The `ComponentView` in `javax.swing.text` is a decorator for the `Component` and adds the functionality of the `View` interface. Another example of a decorator is the relationship between `FilterInputStream` and `BufferedInputStream` of the `java.io` package where the latter adds a buffering feature to the input stream reducing I/O expenses.

Facade

This design pattern belongs to the structural patterns and offers a (often) unified interface to a set of classes and their related interfaces, thus providing simplified accessibility. It also decouples the client code from the underlying subsystem and its possible implementation changes.

This is a widely used pattern that helps in reducing complexity and object coupling. The simplest example may be an object with a method that calls several other classes.

Flyweight

The flyweight pattern, which is a structural pattern, offers the possibility to share one instance of a set of interchangeable instances containing the same information avoiding the memory-intensive expenses of multiple instances.

Proxy

With a proxy the control of an object is moved to an other object that is placed in front of it providing the same interface. So the proxy hides the information on what is happening behind it, while presenting the client with a concrete interface. The proxy design pattern is a structural pattern.

Proxy classes are typically used when the actual targeted object is not ready to accept requests because it is not yet instantiated, finished loading, or located on a remote machine and communication is delayed — so clients may send requests to the proxy which in turn postpones these requests to the buffered object.

An example of the proxy pattern is Java's `Proxy` class located in the `java.lang.reflect` package. It may be used to send received messages to a single object or to a set of further objects, while the sender has knowledge of the proxy only.

Chain of responsibility

Allows one to avoid the direct coupling of a sender and a receiver of a request by sending the request to a chain of objects which may handle the request or pass it further. The involved receivers within the chain are independent and do not know of the other participants, resulting in a loose coupling by the request object that is passed between them. The chain of responsibility pattern belongs to the behavioral patterns.

EventHandlers may be interpreted as uses of the chain of responsibility pattern. (See also the observer pattern.)

Mediator

The mediator controls the communication between objects. The objects communicate to each other via the mediator because it is inserted between the communicating objects and decouples them by providing a centralized communication interface and preventing direct object references. This pattern is a behavioral pattern and helps to reduce the complexity of large sets of interacting objects, by introducing a loose coupling between the interacting objects.

> "The Mediator makes loose coupling possible between objects in a program. It also localizes the behavior that otherwise would be distributed among several objects." [12, page 167]

Memento

This pattern belongs to the group of behavioral patterns and provides the revelation of an object's internal state without violating encapsulation to restore that object to this state at a later point in time.

> "While supporting undo/redo operations in graphical interfaces is one significant use of the Memento pattern, you will also see Mementos used in database transactions. Here they save the state of data in a transaction where it is necessary to restore the data if the transaction fails or is incomplete." [12, page 176]

State

This pattern is a behavioral pattern being used to enable an object to change its behavior depending on its internal state. The object seems to change its class.

Strategy

The strategy offers to change the implementation of behavior by selecting the algorithm independently from the calling clients, and so is a behavioral pattern.

> "Strategy allows you to select one of several algorithms dynamically. These algorithms can be related in an inheritance hierarchy or they can be unrelated as long as they implement a common interface." [12, page 201]

This pattern can be found in the `LayoutManager` of the `java.awt` package with the Container playing the context role. The subclasses of `LayoutManager` use different algorithms for laying out the components. Like the `GridLayoutManager` places its components in a rectangular grid, the `FlowLayoutManager` orders the components much like lines of text in a paragraph.

Template method

The template method pattern is a behavioral pattern as well. It defines the skeleton of an algorithm in an abstract class using a concrete method calling abstract methods. This way it defers the actual implementation to its subclasses assuring that the algorithm's structure is not altered.

A simple example may be given with an abstract class that defines at least two methods, while one is declared abstract and the other method will call it.

```
public abstract class Template {
  public abstract void doInit(); //abstract methods
  public abstract void doPart1();
  public abstract void doPart2();
  public abstract void doFinish();
  public void call() { //algorithm body
    doInit();
    doPart1();
    doPart2();
    doFinish();
  }
}
```

The class `Template` defines a rough procedure or an algorithm and leaves the actual implementation to its subclasses.

Visitor

Similar to the iterator this pattern is a behavioral pattern and allows one to traverse through a set of objects. But instead of returning each object the visitor performs operations on these objects. The visitor pattern allows one to flexibly add additional functionality to objects by the use of double dispatching.

The objects extended by visitors just have to implement an entry method for the visitor. The visitor must provide a `visit()` method to work with the type of the object to be visited performing the desired action.

```
public void accept(Visitor v) {
  v.visit(this); //double dispatching
}
```

2.2.3 Antipatterns

For the sake of completeness I want to mention the term *antipattern*. Antipatterns describe common bad solutions. The knowledge of the most common antipatterns help software developers avoid falling into these pitfalls and help save time and resources.

"The study and application of antipatterns is one of the next frontiers of programming. Antipatterns attempt to determine what mistakes are frequently made, why they are made, and what fixes to the process can prevent them." [40, page 11]

Tate [40] covers the topic broadly and shows procedures to avoid bad software design.

2.2.4 Summary

The idea behind design patterns is to present solutions for common design problems helping the developers not to reinvent the wheel again and again, to assist discussing on a higher level and to help understanding existing software systems. Design patterns are identified design solutions to common recurring problems. Design patterns are not finished and ready to implement solutions but are a description how to solve a class of problems.

"Patterns are more than macro expansions - every time you see a pattern used, it looks a little different. I often say that patterns are half-baked - meaning you always have to finish them yourself and adapt them to your own environment." [19]

Design patterns may be classified by their effect or design level where they are implemented. There are creational design patterns that help to organize the process of object creation, structural design patterns that form structures and assist interactivity within the objects and behavioral design patterns that are concerned with how actions are conducted and how behavior may be managed.

While the design patterns help solve design problems antipatterns refer to common design failures and knowledge of them helps developers avoid pitfalls.

UMLet is designed to allow users to extend its functionality for their needs not only by being written using Java as the programming language. The emphasized use of design patterns helps users that want to extend UMLet understand how it is structured and how the components work together. Design patterns help one to understand the existing design as fast as possible.

Chapter 3

Requirements for lightweight UML tools

Requirements for lightweight UML tools naturally differ from more sophisticated and heavy-weight software suites. Heavy-weight tools focus on backing the whole cycle of software development utilizing some processes like Rational's *Unified Process*, providing round-trip engineering mechanisms for creating source code for different programming languages and vice versa creating models and diagrams from source code. But this requires users to strictly enforce standards and constraints. The users are forced to create exact and standards-conforming designs.

Although strict designs and models may seem to be desirable at the first look, the creation process is complex and time-consuming. In the beginning of a software project it is often necessary to experiment with a design, and the user wants to modify the diagram in a convenient and simple manner. Restrictions in work flows and forced standards make heavy-weight tools unsuitable for design and idea explorations.

Lightweight UML tools are not loaded with tons of features and offer the users the possibility to sketch their diagrams very efficiently by keeping the tool's functions and user interfaces simple. Standards are not enforced restrictively, rather the user is more like being guided to the standard offering more flexibility in the creation of diagrams. The purpose of explorative sketching is to create a design to explain an idea rather than to create a model for implementation.

The requirements for lightweight UML tools are:

3.1 Easy to use

A simple user interface is essential to most applications that require non-trivial user interactions, especially if the software is intended to be used by a non-closed group of unspecialized users.

3.2 Easy to learn

Although user interfaces have a great influence on how easy it is to get into using a software, this requirement concentrates on other scopes like how the application assists the users in fulfilling their tasks. The difference from *Easy to use* is simply in the level of interaction. Here we don't concentrate on how buttons and other user-interface elements are placed or how meaningful icons and illustrations are. The goal is to design the software and workflows in a way that the users find understandable, usable and enjoyable.

UMLet meets this requirement by providing element palettes offering the full-sized elements as they occur and with usage examples.

3.3 Easy to deploy

Simple software deployment is not always an issue. But it is indeed so when the software is used on a greater amount of workstations and version changes of the file format may require upgrades to the software on every single workstation. If the deployment process is simple, it may be sufficient just to drop some files.
It is also desirable to have the possibility to deploy the software on different operating systems and platforms instead of being bound to one single environment. Especially when using software in education, teachers and/or administrators face a wide number of different platforms and operating systems and operating system versions that may need individual configuration.

3.4 Relaxed standards restrictions

Sophisticated UML tools like Rational Rose are very restrictive with UML standards, forcing the users to create strictly correct elements and relations.
Tools intended for easy creation of UML models should allow the creation of diagrams that might not exactly conform to standards: *Explorative sketching* describes the creation of not necessarily exact and /-or completely correct models and diagrams. The goal of explorative sketching is to play and experiment with the model in order to reach a goal iteratively. It is also used to sketch a design and modify it while discussing the model with colleagues and other stakeholders in the requirements analysis phase of the project where simple and fast modification of the diagram and its elements are crucial and exact models are not mandatory.

> "The ultimate goal of modeling is to produce a description of a system at some level of detail. The final model must satisfy various validity constraints to be meaningful. As in any creative process, however, the result is not necessarily produced in a linear fashion. Intermediate products will not satisfy all the validity constraints at every step. In practice, a tool must handle not only semantically valid models, which satisfy the validity constraints, but also syntactically valid models, which satisfy certain construction rules but may violate some validity constraints. Semantically invalid models are not directly usable. Instead they may be thoughtful of

as “work in progress” that represent paths to the final result.” [38, page 125]

“On the other hand, during early stages of design, a developer may not care about the value of a particular property. It might be a value that is not meaningful at a particular stage, for example, visibility when making a domain model. Or the value may be meaningful but the modeler may not have specified it yet, and the developer needs to remember that it still needs to be chosen.” [38, page 125]

Chapter 4

Related work

4.1 Generic

UML [9] was originally developed by Grady Booch, James Rumbaugh, and Ivar Jacobson in 1994. The main reason was to standardize the many existing graphical notations like Booch, OMT, or ERM, and to provide a unified way of describing different software artifacts (classes, components, packages, etc.) arising in different software environments (object-oriented platforms, real-time systems, state machines, etc.). Typical types of UML diagrams are use case, class, or activity diagrams [20]. UML has become a de-facto standard since then. The Object Management Group (OMG), a non-profit industry consortium, is responsible for defining and maintaining the UML language specification [42]. As of early 2006, the various parts of UML are being upgraded to version 2.0.

4.2 Application areas

UML has been applied in many different areas of software engineering. Evans and Wellings [18] report on the application of UML in real-time systems. Kohler et al. [28] use a subset of UML in describing a decentralized production control system. Astesiano and Reggio [3] apply UML in the context of distributed systems; they rely on the language's standardized extension features to adapt it to their specific domain.

UML is also being taught in many software engineering courses; for an experience report please refer to [13]—Hansen et al. [25] discuss the use of tools in lectures of object-oriented modeling and criticize on the tool's common strict standard restrictions. Turner et al. [44] report their experiences of introductory computer science lectures using UML tools.

4.3 Tools

Many tools attempt to provide UML support, e.g., IBM's Rational Rose (http://www.ibm.com/software/rational), Artisan's Real-Time Studio (http://www.artisansw.com), or Borland's Together (http://www.borland.com/together). These tools provide round-trip-engineering capability, i.e., they can produce code stubs from diagrams, diagrams from existing source code, and ways to keep them in sync when one artifact changes. Other tools, e.g., UMLet (http://www.umlet.com) or Violet (http://www.horstmann.com/violet/) focus on the fast creation of UML sketches [11, 6]. An extensive overview of UML tool features is available at http://www.jeckle.de.

See the next but one section below for a short introduction of UMLet's direct competitors. Chapter 6 gets into a detailed comparison of UMLet and Rational Rose.

4.4 User interface design

User interface design [36, 43] is not a strictly quantitative engineering discipline: it requires intuition of users' habits and environments, common sense, an understanding of graphical representations and their effects, and attention to psychological processes during user tasks. An ubiquitous aspect of user interfaces—used widely in any windows-based operating system—are so-called *modal* dialogs. Such dialogs are windows that pop up to request a user input, while freezing input to the rest of the application. These disruptive dialog windows are described by Quan et al. [35] as follows: "Dialog boxes that collect parameters for commands often create ephemeral, unnatural interruptions of a program's normal execution flow." Many developers, too, seem to become aware of this intrusive effect; several well-known applications, like Microsoft's suite of development tools or the Firefox Web browser, for example, now rely on non-modal windows or unobtrusive task bars to provide document search functionality.

For applications with a largely graphical user interface, Moran et al. [33] implemented and refined an alternative user interaction approach—a pen based gesture recognition. While the gesture-based approach requires some initial learning effort, the authors report that users found it understandable and easy to use. Chen et al. [11] apply a whiteboard approach to the creation of UML diagrams in early project stages. Auer et al. [6] describe a text-based user interface to modify the graphical representation of UML diagrams. Several other authors report on possible user interface improvements for UML tools [41, 45, 29].

It is notoriously tricky to quantify how good a user interface is. Users with different background, experience levels, and goals can't possibly agree on a single "best" user interface. Yet several user interface guidelines attempt to provide formal criteria that should at least minimize bad interface choices and provide some cross-platform standardization [2].

Other authors, most notably Raskin [36], attempt to provide metrics to measure the quality or usability of user interfaces. Indeed, a whole family of so-called GOMS methods (goals, operators, methods, selection rules) try to measure usability. For example, KLM-GOMS relies on 6 primitive operations (pressing a key, moving the mouse pointer, dragging the mouse pointer, mental preparation, moving hands, and waiting for command execution), and empirically determined execution times. This paper uses a similar, simplified approach to evaluate how two UML tools perform some fundamental UML design workflows.

4.5 Lightweight UML tools and text-based input

As stated above lightweight UML tools have to be easy to use, easy to learn, easy to deploy and offer relaxed standards restrictions.

4.5.1 ArgoUML

According to the information provided on their web page ArgoUML[1] is the leading open source UML tool. The latest release supports just UML version 1.4. As well as UMLet, it is cross-platform capable through its pure Java implementation. It features diverse output formats to provide interoperability with other tools. A key feature is the the creation of source code from the designed models (Java, C#, C++, PHP). ArgoUML comes with a complex UI featuring many icons, menus, pop-ups, tab-panes, and context depending mouse operations with key-modifiers (using alt- and ctrl-keys).

- Easy to use: The user interface is rather complex. Important information is spread across the screen.
- Easy to learn: Users need substantial effort to use the tool—the manual is more than 400 pages.
- Easy to deploy: The installation package is only 6,5 MByte large. The tool can be deployed very simple.
- Relaxed standards restrictions: Since the tool is aimed at the generation of code, it enforces strict standards restrictions.

The commercial product Poseidon[2] is based on ArgoUML emphasizing the use of DSLs and aims at the model driven software engineering process.

4.5.2 Violet

"Roses are rational, violets are GNU" Mike Godfrey. Violet[3] is designed for educational use and aims at the simple creation of diagrams for export as images and printing. It is easy to learn and use, and cross-platform capable. It does not provide as much UML coverage as UMLet does. The simple user-interface is a result of the small range of functions. Violet may be integrated into web sites as a Java-applet.

- Easy to use: The user interface is very simple. It offers basic drawing assistance.
- Easy to learn: It is very simple and easy to learn ho to draw diagrams.

[1]http://argouml.tigris.org/
[2]http://www.gentleware.com/
[3]http://www.horstmann.com/violet/

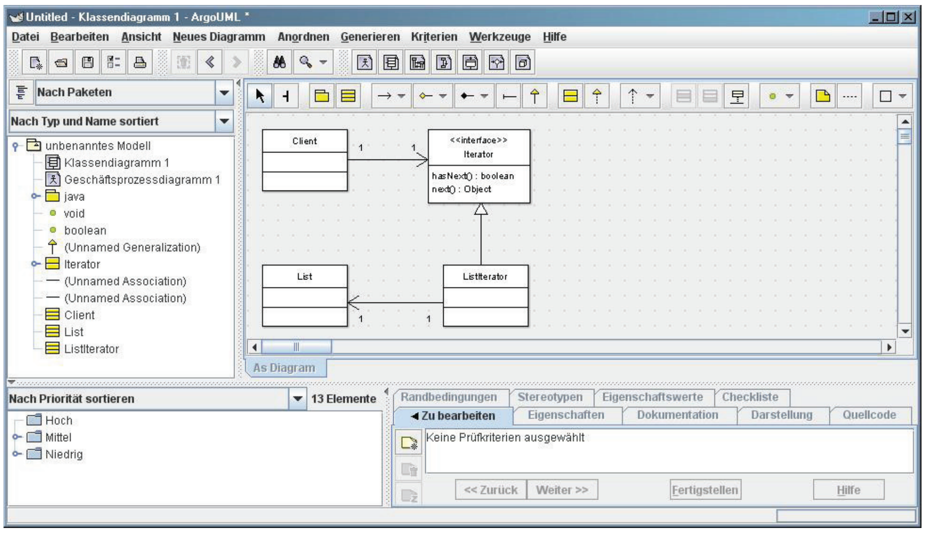


Figure 4.1: ArgoUML

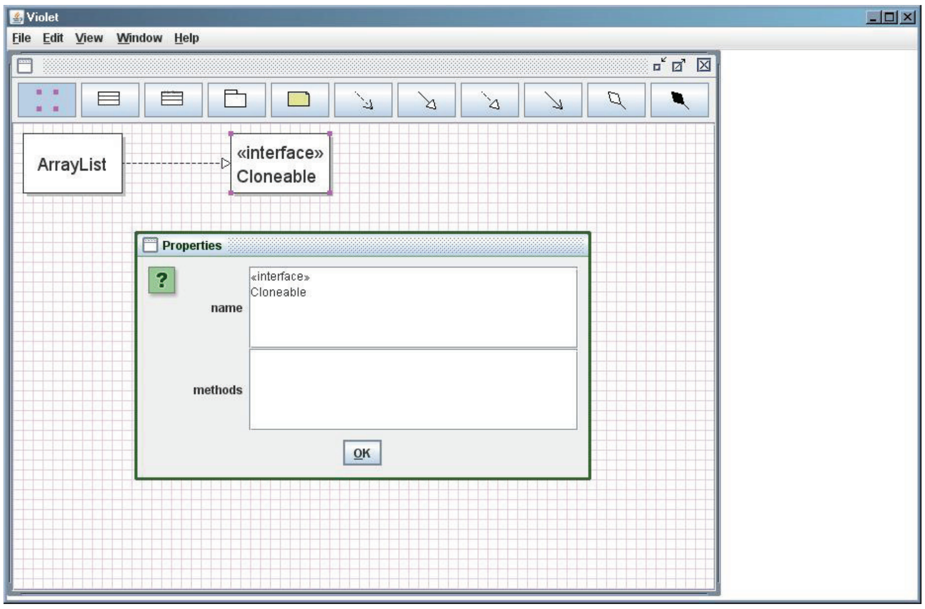


Figure 4.2: Violet

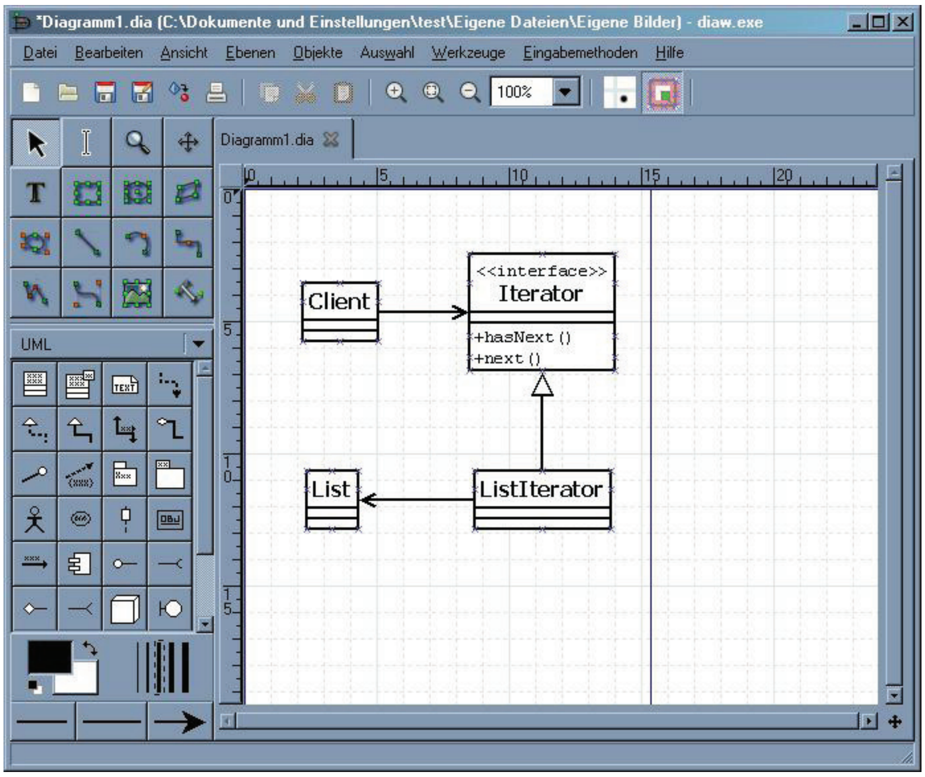


Figure 4.3: Dia

- Easy to deploy: The tool can be deployed and distributed very easy. Its distribution size is only about 300 KByes.
- Relaxed standards restrictions: There is no strict standard enforcement.

4.5.3 Dia

Dia[4] is another well known open source diagramming tool. Like Microsoft's Visio it's aim is at general diagram creation, supporting a wide range of notations. There is special support on entity-relationship diagrams, flow-chart diagrams, and UML. It may be extended by the user by using an XML description (as a subset of SVG) of the shapes. Furthermore it features an interface to Python. Export to various formats eases the interoperability with oder tools. The user interface of Dia utilizes a vast amount of icons, menus and pop-ups. Dia comes as installable binary for Windows, Linux or Mac OS. It can export to a large number of formats including VDX (Microsoft XML for Visio) and DXF (Autodesk Autocad).

- Easy to use: Modeling is interrupted by rather complicated pop-ups for detailed data input.

[4]http://live.gnome.org/Dia

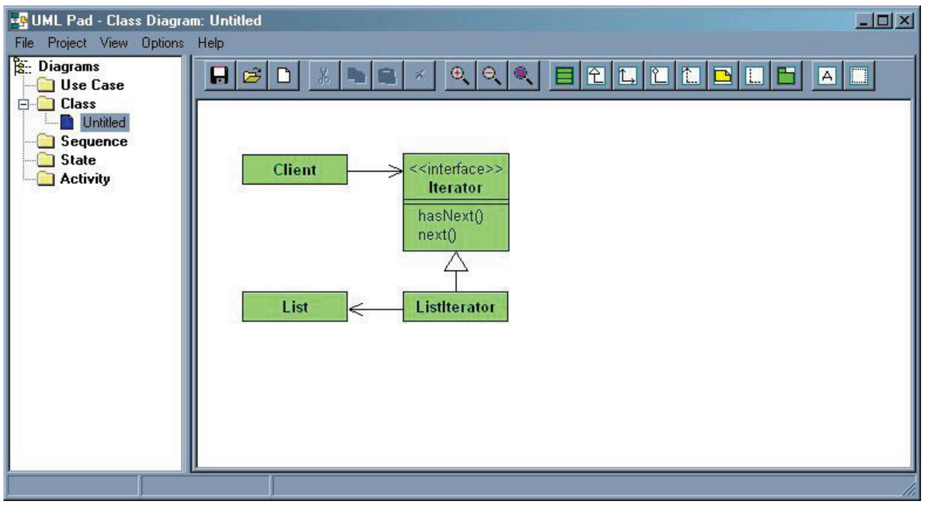


Figure 4.4: UML Pad

- Easy to learn: The tool itself is simple, but the icons are hard to identify.
- Easy to deploy: The deployment process includes a simple installation procedure. The distribution package is about 17.5 MBytes large, which is acceptable.
- Relaxed standards restrictions: There is no strict standard enforcement.

4.5.4 UML Pad

UML Pad[5] is a minimalistic UML drawing tool for Windows. It is possible to export class diagrams as HTML documents. All other diagrams can be exported as images. The user-interface features the essential icons. Object details are specified by pop-ups having tab-panes.

- Easy to use: The user interface is very simple, but there is no drawing assistance like grids or other help for aligning objects.
- Easy to learn: The tool is very simplistic, and therefore very easy to learn.
- Easy to deploy: Deployment is very simple. Just unpack the 1.5 MByte archive and start the tool.
- Relaxed standards restrictions: There is no standard enforcement at all.

4.5.5 Visual Paradigm

Visual Paradigm is a heavy weight modeling suite featuring different model types. Besides UML there is support for business process modeling, database modeling, and diagramming like mind maps, etc. The tool features round-trip engineering—producing code from models, and models from code—for different languages. There are 4 commercial editions and a free community edition. The prices of the commercial editions

[5]http://web.tiscali.it/ggbhome/umlpad/umlpad.htm

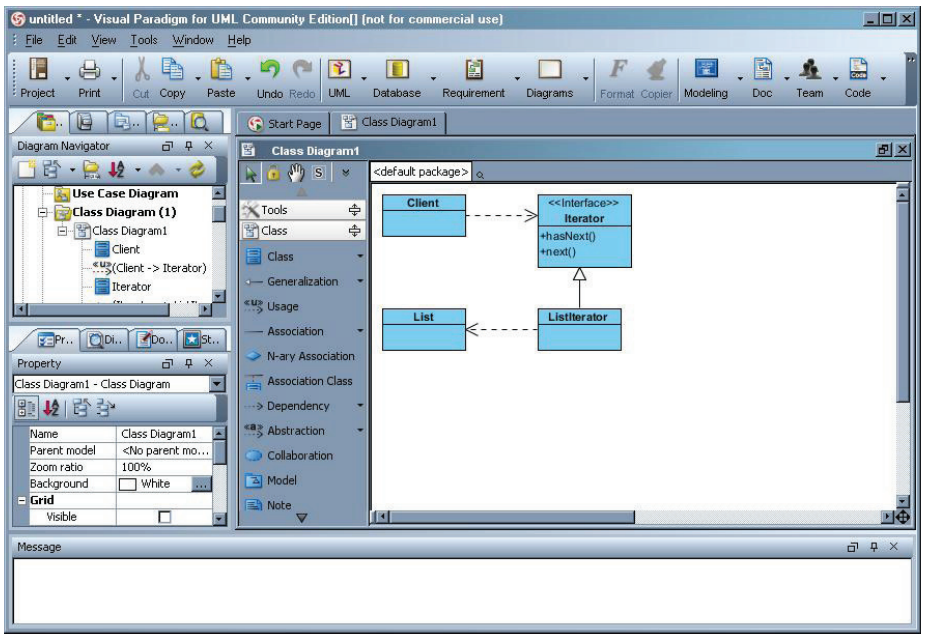


Figure 4.5: Visual Paradigm

are from 99 USD to 1,399 USD. It is available as Java version (cross-platform) and Windows as well as Linux and Solaris. The user-interface is complex and features icons, fancy drop-down menu-lists, pop-up windows, tabbed panes, etc. It features diverse export and import types. It is even possible to import Rational Rose projects, although it is not possible to export to Rational Rose.

- Easy to use: The user interface is very complex—the information is spread over the screen.
- Easy to learn: The software suite is rather complex and features a huge amount of functions.
- Easy to deploy: Deployment requires the user to download about 200 MBytes and about 350 MBytes of space after the installation. Even for the free community edition it is necessary to register and apply a license key.
- Relaxed standards restrictions: There tool enforces strict standards since it aims at the generation of code.

4.5.6 Rational Rose

Rational Rose is one of the leading UML tools used in large-scale software development environments. Rational Rose is no longer sold as a stand alone application, but it is integrated into a number of CASE software products offered by IBM. Rational Rose is not intended to be used as a simple sketching tool—it aims at providing assistance in designing large software systems. It relies on a formal, internal object model that allows one to view the design from different model perspectives. It features round-trip engineering and export to visual documentation formats.

Since Rational Rose is the de-facto industry standard of UML tools, all other serious competitors have to match with it. The more detailed discussion on it's user interface and features can be found in chapter 6.

Chapter 5

UMLet's agile approach

UMLet is focused on providing a simple but powerful user interface avoiding needless pop-ups. It assists the user in an unobtrusive way.

To afford simple and broad distribution UMLet's development focuses on applying and facilitating widely accepted software standards. So UMLet was developed using only Java. Its file format is based on the open XML standards. Furthermore there is no use of any operating system or platform-dependent features. UMLet manages to run without any special configuration or installation routines achieving a high level of flexibility and system independency.

Further concepts are based on usability issues and ergonomics. By abandoning needless windows and pop-ups UMLet provides a smooth and fast way of working. Without interrupting the work flow by diverse menu and dialog interactions, working with UMLet turns out to be simple and goal-oriented. Some diagram types feature a simple yet powerful grammar completely avoiding time-intensive or complex mouse interactions.

UMLet may be used as a plug-in for the Eclipse software development system allowing it to combine the requirements analysis and documentation of the to be developed software system with the source code management. The systems analysts can share their analysis results directly on a common level using UML and other diagram types. This combination and aggregation allows the manipulation of documentation files and source code in parallel within a coherent environment. Furthermore the documentation files may be shared with other development colleagues automatically allowing them to get into already existing modules more quickly.

To allow users to easily extend UMLet's range of elements and diagram types UMLet features its *custom elements*. These custom elements may be developed, extended and modified within UMLet at runtime using the Java language. These elements may be easily shared with colleagues as well.

Especially for making the introduction into working with UMLet easy, it features an element palette panel with the original sized elements, instead of a set of abstract and tiny icons. So users do not need to try several icons in order to find the right one, instead they can directly identify the correct element. Furthermore these palettes may be rearranged to match the user's needs.

This chapter will discuss UMLet's internal concepts and ideas in more detail.

5.1 Simple standard file format

"XML has emerged as a powerful and easy way to save data in files. Because it is a standard, XML enables you to save data in a form that can be accessed by applications other than the ones that created the data. XML software, much of it free, enables you to access the data in XML documents using standard application programming interfaces (APIs)." [24, page 5]

XML is the acronym for eXtensible Markup Language. It is a domain independent meta-markup language. The term *meta-markup* describes the fact that it is not a language to represent data, but it is a tool to describe the languages that handle specific data. XML is used to define the low level language syntax: How element data is distinguished from its content, what attributes may be attached and how. So XML defines the syntax and semantics of a language.

"XML, the eXtensible Markup Language, is not actually a language in its own right. It is a metalanguage used to construct other languages. XML is used to create structured, self-describing documents that conform to a set of rules created for each specific language. XML provides the basis for a wide variety of industry- and discipline-specific languages. Examples include Mathematical Markup Language (MathML), Electronic Business XML (ebXML), and Voice Markup Language (VXML)." [21, page 38]

The big advantage to static mark up languages like HTML is that the set of tags is infinite. Developers can define their tags to fit their needs. Since HTML is defined for document viewing it is limited to format and layout human readable text. XML is built to break these limitations and aid representing any data. The only constraint that is given is that the data to be contained must be represented by characters.

The fact that XML files are human-readable eases the development and debugging of applications, since no special file viewer is needed and any text viewing tool may be used.

"Domain-specific tagging has a number of advantages, not the least of which is that its easier for a human to read the source code to determine what the author intended." [26, page 5]

"In XML, both markup and content contribute to the information value of the document. The markup enables computer programs to determine the functions and boundaries of document parts. The content (regular text) is what's important to the reader, but it needs to be presented in a meaningful way. XML helps the computer format the document to make it more comprehensible to humans." [37, page 7]

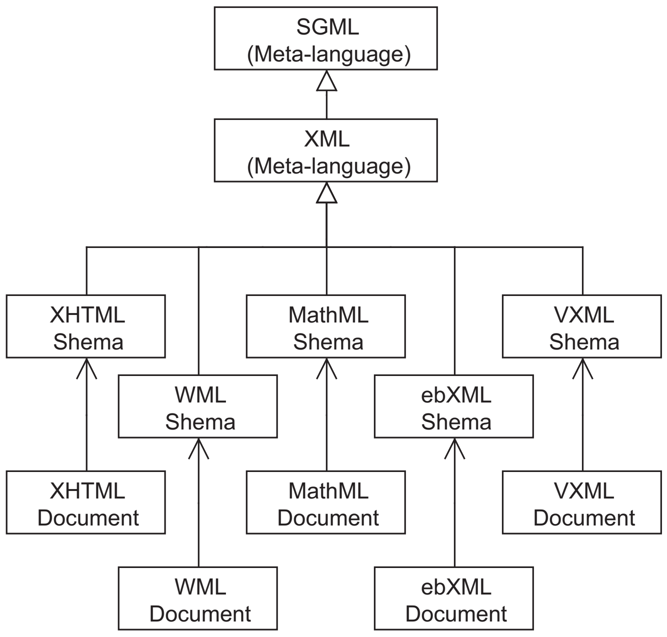


Figure 5.1: XML language hierarchy [21, page 39]

```
<?xml version="1.0" encoding="UTF-8"?>
<umlet_diagram>
    <element>
        <type>com.umlet.element.custom.Component</type>
            <coordinates>
                <x>10</x>
                <y>100</y>
                <w>160</w>
                <h>80</h>
            </coordinates>
            <panel_attributes>Component</panel_attributes>
            <additional_attributes></additional_attributes>
    </element>
    <element>
        <type>com.umlet.element.custom.ActiveClass</type>
            <coordinates>
                <x>10</x>
                <y>20</y>
                <w>160</w>
                <h>60</h>
            </coordinates>
            <panel_attributes>ActiveClass</panel_attributes>
            <additional_attributes></additional_attributes>
    </element>
</umlet_diagram>
```

The example above shows an UMLet file containing the two elements `Component` and `ActiveClass` with their coordinates within the diagram and their attributes. Furthermore the example demonstrates the connection between tag and data that allows easy interpretation.

The string representation also assists data transfer between computer systems over networks and via different platforms and operating systems. Developers do not need to worry about byte representation issues like byte and word ordering but can rely on the data and its string representation.

Using a parser is the best practice to use when reading the data contained in XML files.

> "An XML parser is a software component that can read and (in most cases) validate any XML document. A parser makes data contained in an XML data structure available to the application that needs to use it." [21, page 44]

There are the two common ways to parse XML files: DOM and SAX.

- DOM:
  DOM stands for Document Object Model. A parser following the DOM idea reads the whole document in at once and constructs an element tree in memory. This method is suitable for relatively small documents only, because of its lack of speed and mentionable use of memory resources. The advantage compared

to SAX is the possibility to modify the element tree, add and remove elements in a relatively simple way.

> "DOM (document object model): A specification that defines the structure of a document as a collection of objects and how the document can be accessed and manipulated. A document object model is an API (application programming interface), which describes how programs actually interpret the structure and contents of a document." [37, page 256]

- SAX:

> "SAX (Simple API for XML): An event-driven application programming interface for manipulating XML documents with Java. The API describes a flat document model (no object hierarchy or inheritance) that then allows for quick document processing." [37, page 268]

> "Using SAX, the parser reads in the XML data source and makes callbacks to its client application whenever it encounters a distinct section of the XML document. For example, a SAX event is fired whenever the end of an XML element has been encountered. The event includes the name of the element that has just ended." Gabrick and Weiss [21, page 44]

UMLet uses the second procedure to retrieve the XML data. Every time an XML element is detected the `endElement()`-method of the `XMLContentHandler` is called which associates the element type with its data (Figure: 5.2).

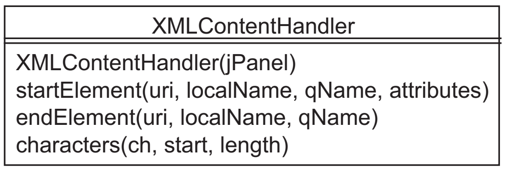

XMLContentHandler
XMLContentHandler(jPanel)
startElement(uri, localName, qName, attributes)
endElement(uri, localName, qName)
characters(ch, start, length)

Figure 5.2: XMLContentHandler class

5.2 Independence from platforms and operating systems

> "Platform independence, the separation of an application from the platform on which it runs, is the essential characteristic of Java. With Java, software developers can write a program once, and it will run on any platform that has an implementation of the Java virtual machine." [15, page 1612]

UMLet's platform independence is achieved by using only Java for development. No platform-dependent features are used. So users are not bound to a specific operating system like Windows or Linux. All that is needed to run UMLet is a Java virtual machine implementation. Java runtime environments are available from Sun Microsystems for a number of operating systems including Microsoft Windows, Sun Solaris, Linux and Apple. Other operating systems are supported through Java implementations by IBM and other vendors.

> "Why has Java taken over so quickly? The short answer is found in its platform independence and potential to turn the Web into a much more dynamic and interactive environment – something that is badly needed." [34, page 3]
>
> "In the old days of computer languages [...], programs were designed to run under a single operating system, more or less, and the name of the game was to create programs that could run as fast as possible. Almost over night, the World Wide Web and Java have changed this notion of operating system-based language environments to platform-independent network-driven languages and systems." [34, page 4]

This independence from platforms and operating systems allows UMLet to be used in heterogeneous fast changing environments like large development sites or universities with a vast amount of different computer systems.

Since UMLet's installation process does not even make use of any particular installation tools, no special operating system-dependent distribution files have to be provided. The user has just to un-zip the distribution package that he downloaded and start the contained JAR file. There are also no specific settings or configurations to be taken. So users do not have to be supported with any installation and configuration aid.

UMLet was (and still is) developed for educational uses:

> "When teaching tool-supported software development concepts, both concepts and tools have to be taught. Using heavyweight tools can both distract students from the main educational goals [1, 8] and cause substantial costs - even if UML tool providers often do not charge license fees to universities." [7]

The installation packages of heavyweight tools often have 200 megabytes or even exceed that size:
IBM Rational Rose Enterprise Evaluation V2003.06.15 for Windows 2000, NT, XP English Japanese C82XPML.exe (354 MB) Date: October, 8th 2005.

The installation process on the computer environment in classrooms may be hard to automate due to interactive installation procedures. After successful deployment one has to deal with license keys, eventually with a license server and with license expiry dates.

UMLet's distribution, installation and configuration process is kept as small and simple as possible so that students can manage to set-up UMLet on their own on their home computers. And since UMLet focuses on the basic features needed for UML creation, and is not overloaded with tons of features and functions the amount of required computing performance is fairly small. So there is no need for up-to-date high-end computers, which accommodates students who may not have such advanced hardware systems at home.

Another important point of development was the licensing issue. Using commercially distributed software packages often includes the burden of obtaining special licenses for educational use that have to be spread to the students. Leaving the distribution of the software package aside, delivering the license keys can require substantial effort too. To prevent other persons than the participating students receiving the software's license keys, these keys have to provided on a password-protected FTP server (or a password-protected area of the web server) or by copying CDs.

UMLet is distributed as open source and protected by the GNU license only. It may be freely copied and modified for individual needs. There are no license keys, serial numbers, or activation obstacles.

5.3 Duality: Application / Plug-in

UMLet may also be used as a plug-in for the widespread software development environment Eclipse. Eclipse is developed in the Java language but is not bound to it. So there is support for developing software in other languages like C and C++, C#, COBOL, Python, Haskell, OCaml and others.

Eclipse was donated to the open source community by IBM. Like SUN Microsystems does with OpenOffice and its StarOffice, IBM offers a commercial version of Eclipse called WSAD (WebSphere Application Developer) which will be replaced by the Rational Developer (RAD) in the near future. The difference between these versions is the delivered features. Eclipse is delivered featuring about 90 plug-ins, WSAD offers about 500 to 700 plug-ins providing the developers with functionality to develop web and database applications.

> "As long ago as 1988, OTI developed a collaborative development environment for Smalltalk called ENVY, which was later licensed to IBM under the name Visual Age. What followed was the development of Visual Age for Java, but this was still implemented in Smalltalk. Now, OTI has started the next generation of development tools with Eclipse. Of course, we find many of the design elements of Visual Age in Eclipse. The difference is, however, that Eclipse is implemented in Java and that it features a much more open architecture than Visual Age." [14, page X]

> "Eclipse is an open source community whose projects are focused on providing an extensible development platform and application frameworks for building software. Eclipse provides extensible tools and frameworks that span the software development lifecycle, including support for modeling, language development environments for Java, C/C++ and others, testing and performance, business intelligence, rich client applications and embedded development. A large, vibrant ecosystem of major technology vendors, innovative start-ups, universities and research institutions and individuals extend, complement and support the Eclipse Platform." [Online: www.eclipse.org]

By copying or deflating the UMLet distribution package into Eclipse's `plugins` directory the installation process is already done. UMLet may be invoked by either double clicking on an already existing UMLet document within the actual Eclipse project or by starting UMLet's "New file"-wizard.

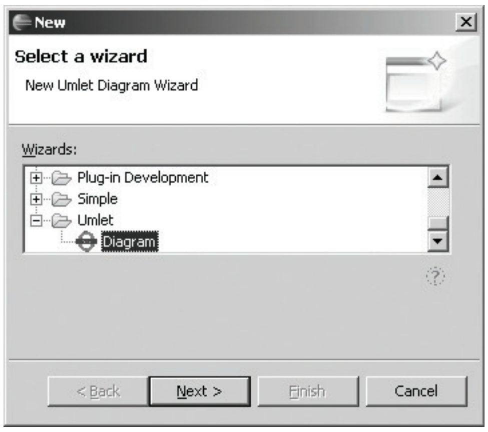


Figure 5.3: New file wizard

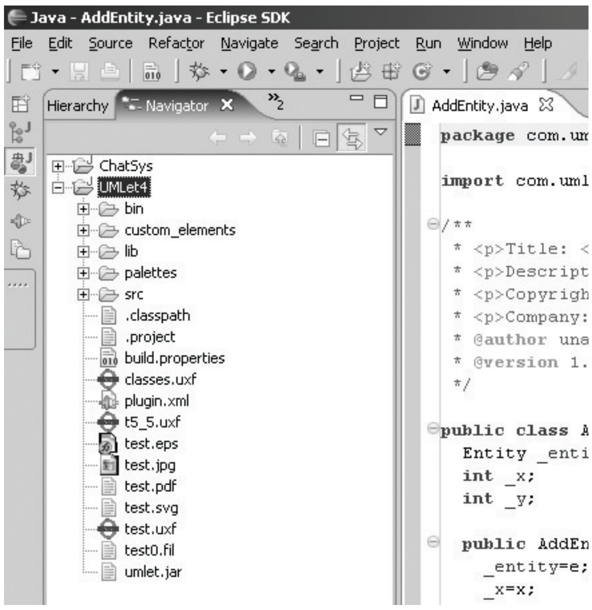
The advantage of using UMLet as a plug-in within Eclipse is based on the integration of the documentation into the Eclipse project. When developing software in a team of software developers, usually a source code versioning system like CVS (Concurrent Versioning System) is used. When *checking in* the project, all team members also receive the documentation done with UMLet.

UMLet appears as an editor within Eclipse allowing one to create and modify the diagrams in common with the source code documents, while still providing full feature support for all stand-alone functionality like exporting to the publishing formats and custom element creation.

5.4 Transparency

UMLet is designed using the Java AWT Graphics (`java.awt.Graphics`) framework for drawing the elements. There is no distinction between drawing on the screen or in a file. The output device is irrelevant. For the drawing process it makes no difference what medium is written to, indeed the `paint()`-Method of the elements have no knowledge of anything else than the `Graphics` class they write to. When exporting the diagram for example to a JPEG file, UMLet just calls the `paint()`-Method of all elements contained in the drawing panel.

```
public class InitialState extends Entity {
  public void paint(Graphics g) {
    g.fillOval(0,0,this.getWidth(),this.getHeight());
  }
  public int getPossibleResizeDirections() {return 0;}
}
```



```
package com.um

import com.uml

/**
 * <p>Title: <
 * <p>Descript
 * <p>Copyrigh
 * <p>Company:
 * @author una
 * @version 1.
 */

public class A
  Entity _enti
  int _x;
  int _y;

  public AddEn
    _entity=e;
    _x=x;
```

Figure 5.4: UMLet files integrated in an Eclipse project

5.5 Easy expandability (custom palettes and dynamic custom elements)

Instead of using schematized icons for the elements, like most other tools do, UMLet utilizes an element palette with graphically identical elements. Users can immediately identify the objects. To be more flexible UMLet supports more than one element palette. These palettes are usually categorized by the use case. There are palettes for building class diagrams, activity diagrams, sequence diagrams and more. Users may also rearrange the palettes to match their needs and speed up working. The desired palette may be chosen using a drop down box in the menu bar.

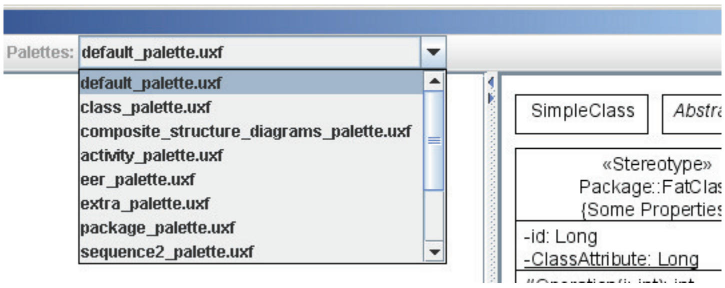
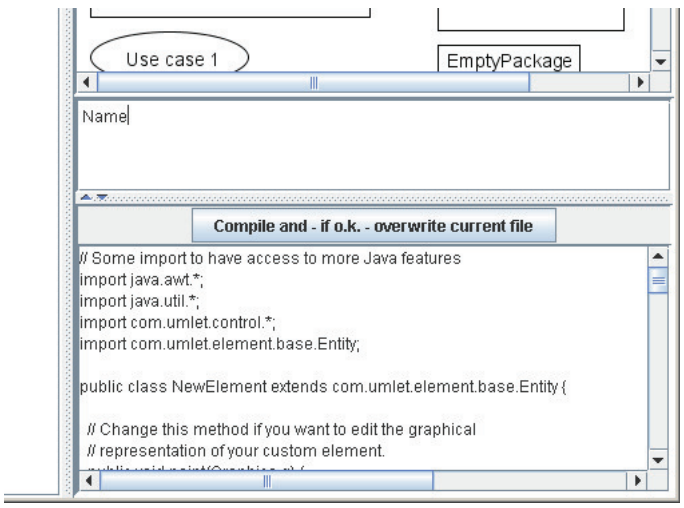


Figure 5.5: Palette drop down box

Adding new palettes to the list of already existing ones is done by copying/saving the new palette to the palettes directory and adding the `_palette.uxf` suffix to the filename like: `sequence_palette.uxf`

A main feature that distinguishes UMLet from other UML tools is the possibility to integrate user made elements. The so called *Custom Elements* feature allows users to extend UMLet's set of entities and it even allows the creation of entirely new diagram types. These custom elements may be implemented using the built in Java compiler. A special reserved text-panel in UMLet allows the modification of the element's Java code (see Figure: 5.6).



```
// Some import to have access to more Java features
import java.awt.*;
import java.util.*;
import com.umlet.control.*;
import com.umlet.element.base.Entity;

public class NewElement extends com.umlet.element.base.Entity {

  // Change this method if you want to edit the graphical
  // representation of your custom element.
```

Figure 5.6: Source code frame

A new custom element may be created by selecting "New Element" from the custom elements drop down box. The source code of the newly-created element is a well-commented basic implementation of an UMLet-Entity, providing examples for drawing the entity, defining the docking/sticking polygon and attribute handling. Since custom elements are derived from a superclass that implements all needed backend functionality like loading, saving, etc. the user may concentrate on the element creation.

For the sample code of a new custom element see the appendix.

Already-existing custom elements may be added to the current diagram using the drop down box.

Sharing custom elements and palettes is as simple as sending the source files of the elements (`(UMLet installation directory)/custom_elements`) and the palette files (`(UMLet installation directory)/palettes`) respectively by email or burning them on a CD.

UMLet realizes this feature with a built-in Java compiler. The source code is saved to a file and compiled at runtime by clicking the compile button in the source code panel of UMLet. Since the source code of the elements are stored to the above-mentioned directory, users may also edit the source code with what ever editor they want, although developing the element within UMLet has the advantage of seeing the changes immediately. The affected elements that are currently edited are marked with an *in progress...* message to easily identify these elements.

After successful compilation of the custom element, loading the class code from the file system and instantiating the object, all instances within the diagram and if so also within the palettes are exchanged with the new instance. The runtime class loading from the file system is done by the `FileClassLoader`. It implements loading the class data from the file system if it is not found in the system cache, which is the case when loading the class for the first time. The instantiation of the class is done by the `CustomElementLoader` which receives the class data from the `FileClassLoader`.

5.6 Unobtrusiveness

A design concept that has big weight is that interaction should be unobtrusive. UMLet does not bother the users with a vast number of windows and pop-ups. Instead UMLet proposes only few possibilities to enter data. The main input is done in the attributes panel.

As described in the following section UMLet provides a schematized way of creating and modifying diagrams and elements, not compelling the user to handle differing data input varieties. These differing data inputs are normally implied by the way conventional user interfaces are built.

But there are still pop-ups in use. The built-in Java compiler utilizes a pop-up window to show error messages that occurred during the compilation of a custom element, either when opening a file and the custom elements are not compiled yet or when the compilation process was manually started. Also for naming a new custom element in the element creation phase UMLet opens a modal dialog box.

5.7 Fast diagram creation

The design decision to avoid pop-ups and diverse configuration windows and only concentrate on a single attribute window opened a way of modifying the elements in a compact and quick manner.

As previously mentioned, conventionally designed user interfaces for data input imply some restrictions which UMLet solves in an elegant way by introducing its textual element description:

- Using “copy and paste” to duplicate parts or the full attribute set of an entity.
- Changing the specific type of a relation.
- Extending and modifying elements using syntax instead of options represented by a complex user interface.

The elements offer an easy to understand syntax of how to modify their characteristics. As an example the class element is represented in its basic canonical form showing its name only surrounded by a rectangle. This is done by simply typing the class name into the attributes window.

SimpleClass

Figure 5.7: A simple class

By modifying the textual attributes only the class may show additional information like variables and methods. The class name, variables and methods are separated by adding a new line consisting just of the string "--". Special properties, like defining a method or the class as being abstract is done by surrounding the text line defining the name of the method or class by the slash character ("/").

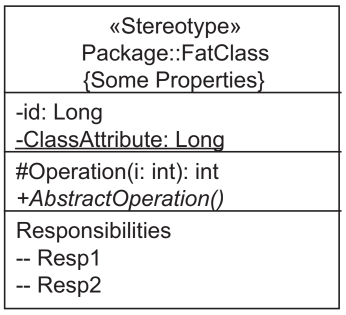


Figure 5.8: A full blown class

It is even possible to change the type of relations by just modifying the attributes. To change an inheritance to a uses relation all that has to be done is just removing a "<" from the definition string.

Allowing copy and paste to and from the element's attributes is a great help when creating a number of similar elements.

UMLet's approach lets the users delay their decision as to whether the elements should be full blown or reduced to basic sizes, supporting them to easily modify the elements at a later point in time. So the users can concentrate on the coarse diagram creation and refine it just on demand. Also the users are assisted by the schematized way of modifying elements. More complex elements are edited in the same way and since the palettes contain examples of how the elements may be used users get into using UMLet very quickly.

Creating a sequence diagram the conservative way by arranging the elements by hand is a time consuming process. UMLet offers a way to create sequence diagrams in a very fast and simple way by utilizing a simple yet powerful syntax allowing to create, modify and rearrange the diagram in very short time:

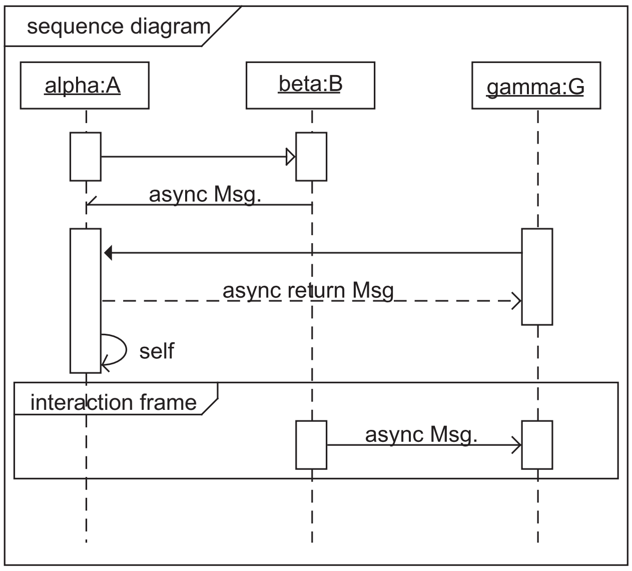


Figure 5.9: Sequence diagram

The diagram of figure 5.9 is defined by the following properties text:

```
title:sequence diagram
_alpha:A_|_beta:B_|_gamma:G_
1->>2:1,2
2-/>1:async Msg.
3->>>1:1,3
1.>3:1,3:async return Msg
1->1:1:self
iframe{:interaction frame
2->3:2,3:async Msg.
iframe}
```

Every attribute text line corresponds to a line in the diagram. The second line defines the involved objects allowing UMLet to calculate the needed width of the entire diagram. Method calls and object activities are stated by the number of their definition. So the second line means a synchronous message from the first object (1[1]) to the second object (2) and stating an activity of both objects. Line three defines an asynchronous message while the objects are inactive. A self call is stated in the same way but that the destination object is equal to the source object. Every message may

[1]The numbers in brackets correspond to the numbers in the textual representation of the sequence diagram in order of their appearance.

be named while the three parts of a message are divided by the colon (“`:`”).

Interaction frames that can show activities conditioned by some constraints may be added by surrounding the desired activities with **`iframe`**{: and **`iframe`**} marks.

5.8 Interoperability

To grant a maximum of interoperability with standard publishing software UMLet is capable of exporting the created diagrams into diverse publishing formats. Up to now UMLet supports exporting to:

- JPG: The broadly used lossy raster images file format developed by the Joint Photographic Experts Group.
- SVG: The Scalable Vector Graphics format – an open standards file format by the W3C, is based on XML and is capable of describing two dimensional vector graphics.
- PDF: The Portable Document Format developed by Adobe Systems is intended for representing documents independently of platforms, operating systems and the software the document was created with.
- EPS: EPS is an extension of the by Adobe Systems developed PostScript file format and is used for PostScript graphics files that are to be embedded into other documents.

So the diagrams created with UMLet may be easily imported into standard text processing software and / or used on web pages.

> “But many times parts of the models must be inserted in Word documents, processed with Latex, or just pasted as low-resolution bitmaps to a PowerPoint presentation. After all, UML is all about communication. However, few UML tools currently support a wide range of file formats. Often one has to rely on additional third party tools like Acrobat to process UML diagrams outside their creation tool; few tools natively support high-quality publishing formats. Often the only way to share diagrams is to print them.” [7]

5.9 Educational use

The concepts presented in this chapter are the basis on which UMLet is developed. The fact that UMLet is written in Java not only supports software developers, but it also helps to integrate UMLet in heterogeneous computer system environments like those found at universities and schools.

- UMLet’s concept of expandability allows the adaption of graphical notations that may be missing for the purpose of the course.
- The intuitive user interface of UMLet allows students to concentrate on solving their exercises and not having to struggle with the application wasting time with badly-designed dialogs and inscrutable help and documentation files.

- The simple installation process of the small distribution files prevents setup frustration.
- Utilizing Java allows application deployment on different operating systems and platforms. So the students are not bound to a specific operating system in class rooms but may install UMLet on their home computers easily.
- Providing different common publishing file formats allows the students to easily process and integrate the created diagrams in their preferred publishing applications.

> "For most students, tools are the real stuff. It is the tool that helps her or him writing, compiling, and debugging programs. It is the tool that makes application frameworks accessible and usable. It is the tool that saves time. It is the tool that is fun to play with. So, learning by tool usage is highly motivating for students." [23]

5.10 Trade-offs

After all the positive features UMLet also has to deal with some trade-offs.

UMLet may suffer from problems of the actually used Java virtual machine. There may be performance issues, when comparing to natively compiled applications. Although modern Java compilers and virtual machines using hot spot optimizers reduce this problem there are still some performance issues. UMLet may behave a little slow in some situations, since SUN's actual AWT (Abstract Windowing Toolkit) and SWING implementations, that are used for implementing the graphical user interface, are not very fast.

But this issue may be solved using a native compiler producing platform dependent executables. This option is left to the users if they want to, since UMLet loses its platform independency by this procedure.

Using palettes instead of small icons implies more space consumption. To reduce this problem UMLet allows one to minimize the palette panel to gain more space on the drawing panel.

5.11 Summary

- UMLet features an easy and unobtrusive user interface avoiding unnecessary windows and pop-ups and providing powerful and fast element creation and modification.
- Its expandability is achieved by the fact that UMLet is available as open source and its custom element feature of allowing flexible element creation using Java at runtime.
- UMLet features a high level of platform and operating system independence by the use of Java for implementation and distribution without the use of system specific installation procedures and an open standards conformed XML file format.

- Users may use and modify UMLet without any constraints, because UMLet is published under the GPL and available as open source.
- UMLet's interoperability with standard desktop software is achieved by allowing one to export the diagrams into a variety of standard publishing formats.
- UMLet may be used as a plug-in in Eclipse integrating documentation into the Eclipse project.
- Simple implementation of element drawing methods allows transparent usage on screen and file streams.
- Focus on educational use issues.

Chapter 6

Comparison: UMLet versus Rational Rose

To prove the advantages of UMLet this chapter will compare UMLet against Rational Rose in its latest version. Section one will discuss some user interface problems that Rational Rose suffers. The later sections present quantitative comparison of both tools featuring a simplified GOMS method by counting and comparing the user interactions needed to execute defined use-cases.

6.1 Usability

This section reflects on some user interface issues users run into when using Rational Rose, especially when using it the first time, and discusses how UMLet approaches these problems.

6.1.1 Element selection

Rational Rose provides access to its elements via a toolbox represented by rather abstract icons. This toolbox can be modified and may hold the following elements:

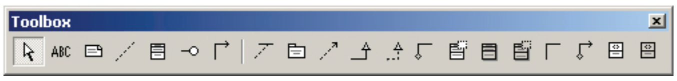


Figure 6.1: Rational Rose Toolbar featuring icons

These icons are hard to interpret without prior knowledge. Rational Rose offers help with small tool-tips – but searching for the right icon encumbers a quick and smooth working process. Alternatively elements may be chosen via the menu by selecting "Tools->Create" and the desired element. This menu is bound to and derived from the actual diagram type. So it is not possible to mix up elements from different types of diagrams. Since Rational Rose concentrates on the model and assists the user to transform the created model into source code and vice versa the limitation may be desired. But for sketching a design and the ideas behind it, this limitation may not be ideal.

Comparing Rational Rose's toolbox containing icons to UMLet's approach using real size elements in its palettes shows UMLet's advantage at one glance: The users don't have to search for the element from a set of abstract and tiny icons or can not get stuck with poorly descriptive element names but simply choose at first sight the desired element from the palette.

> "Using the UML doesn't necessarily imply developing documents or feeding a complex CASE tool. Many people draw UML diagrams on whiteboards only during a meeting to help communicate their ideas." Fowler [20, page 29]

6.1.2 Diagram creation

Since the class diagram is the most often used UML diagram type, UMLet offers the major element types of a class diagram in its default palette which is displayed in the palette panel at startup. So the users can start right away without having to do any prerequisite configuration or initialization steps like defining diagram types, diagram names or anything else.

To create a new class diagram the user has to follow the following steps:

1. Right-click the Logical View entry in the browser.
2. Select New->Class diagram from the shortcut menu.
3. Enter the name of the new diagram.
4. Double-click the diagram in the browser to open it.

After creating a class in the diagram drawing panel Rational Rose prompts the user for the name of the class and offers a vast number of known class names in a selection box. So it's easy to integrate classes of the Java-API.

When adding attributes or operations Rational Rose adds an item with a default name and allows the user to rename the item. Confirmation of the entered name with the "Enter" key adds a new attribute with another default name. The only way to stop the process of adding attributes/operations is by making an action with the mouse, either by clicking a button, menu or simply on an empty space. This uncommon behavior may appear annoying to new users, while it may be handy when creating more than a handful of attributes/operations is desired (see figure 6.2).The type of attributes can be chosen from a selection box which also offers all known types making it a little difficult to select the desired type because of the huge amount of listed items (see figure 6.3). The initial value and visibility options can be defined here as well.

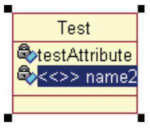


Figure 6.2: Adding new attributes to a class in Rational Rose

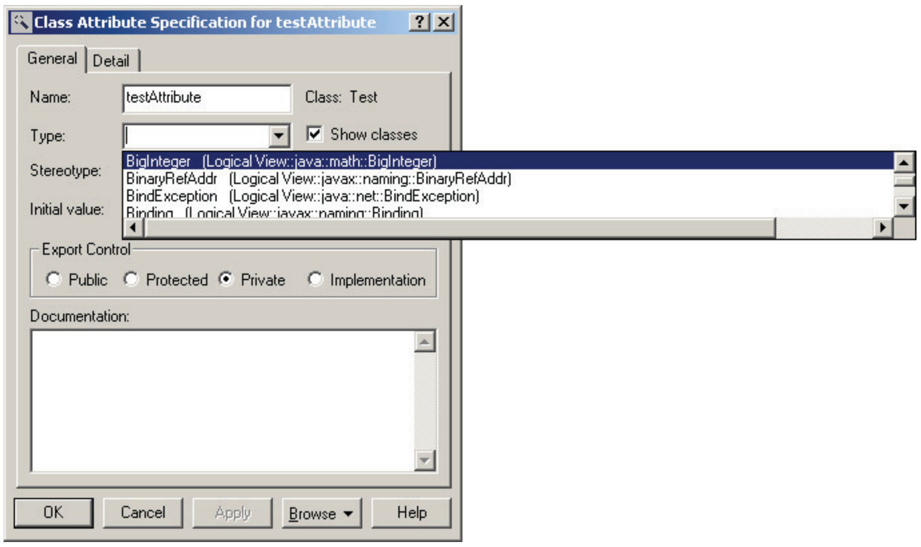

Class Attribute Specification for testAttribute
General
Detail
Name:
testAttribute
Class: Test
Type:
Show classes
Stereotype:
Initial value:
BigInteger (Logical View::java::math::BigInteger)
BinaryRefAddr (Logical View::javax::naming::BinaryRefAddr)
BindException (Logical View::java::net::BindException)
Binding (Logical View::javax::naming::Binding)
Export Control
Public
Protected
Private
Implementation
Documentation:
OK
Cancel
Apply
Browse
Help

Figure 6.3: Attribute type selection dialog

Alternatively Rational Rose offers a *Class Creation Wizard* which guides the user through the class creation process. Unfortunately this wizard only allows one to specify the basic parameters of the class like if it is a super class, or derived from an existing class. No attributes, operations or any other specific details may be declared.

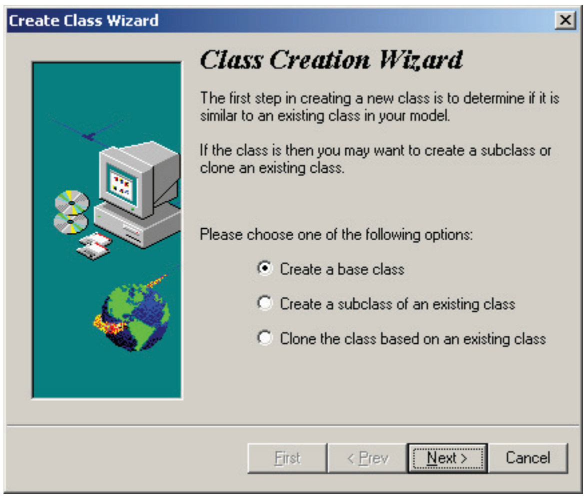
6.1.3 Model types

Rational Rose allows the user to select from a set of predefined model types. There are models for diverse Java versions and other environments. Depending on the selected model type Rational Rose provides class types and attribute types at complying places. This feature enables the users to work with elements already present in the selected environment.

6.1.4 Deleting elements

Deleting an element is not as intuitively solved as one could expect. Since Rational Rose features a model rather than just a diagram, the element to be deleted has to be removed from the model to avoid errors. Rational Rose presents the model in a browser–like window where the entities are listed (see figure 6.5). Elements and associations cannot be removed using the "DEL" key but must be removed from the model using "Ctrl–D" or using the context menu "Edit->Delete-from-Model" or by deleting the element in the model browser.

Contrastingly generalizations cannot be deleted in the browser. Also deleting a generalization from the diagram does not remove it from the model. The user has to open the dialog of the class that is declared as the specialized one by the generalization, then switch to the tab "Relations" and remove the corresponding relation. This



Create Class Wizard

Class Creation Wizard

The first step in creating a new class is to determine if it is similar to an existing class in your model.

If the class is then you may want to create a subclass or clone an existing class.

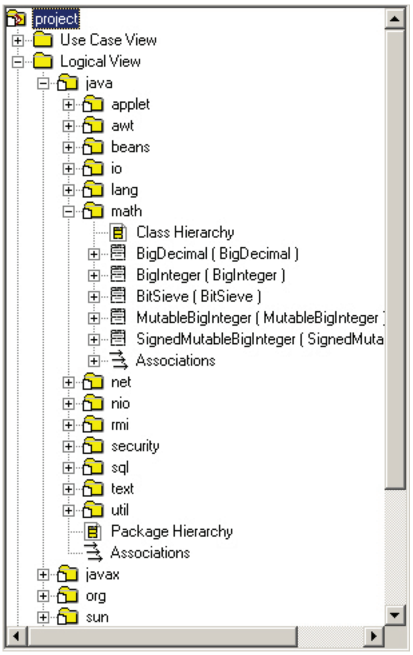
Please choose one of the following options:

[✓ Create a base class]

Create a subclass of an existing class

Clone the class based on an existing class

Figure 6.4: Rational Rose's Class Creation Wizard



```
project
+ Use Case View
- Logical View
  - java
    + applet
    + awt
    + beans
    + io
    + lang
    - math
        Class Hierarchy
      + BigDecimal ( BigDecimal )
      + BigInteger ( BigInteger )
      + BitSieve ( BitSieve )
      + MutableBigInteger ( MutableBigInteger )
      + SignedMutableBigInteger ( SignedMuta
      + Associations
    + net
    + nio
    + rmi
    + security
    + sql
    + text
    + util
      Package Hierarchy
      Associations
  + javax
  + org
  + sun
```

Figure 6.5: Model Browser

is annoying to the user, a time-consuming process and very error-prone if more than one relation exists.

6.1.5 Duplicating elements

Also duplicating elements can not be done by simply selecting the elements and applying any common copy and paste methods. Although the elements are visually duplicated in the diagram, they are not duplicated in Rational Rose's model, but instead these instances refer to the same element in the model. If the user modifies the copied element in the diagram, the original element is changed as well.

This behavior is obscure and not intuitive, which is a pain for users that are new to Rational Rose.

6.1.6 More issues

A handy feature of Rational Rose is its auto layout function which rearranges the selected elements automatically. It also resizes the elements so their content, the names of attributes and operations, fit into the frame. The result is a tidy diagram.

Rational Rose shows an unpleasant undo functionality. Surprisingly the insertion or modification of elements and dependencies cannot be undone within Rational Rose. The creation of an element cannot be undone, but has to be deleted. Also the alteration of the name of an attribute or operation can not be undone, but it has to be done by hand. UMLet is way more flexible and user oriented.

Rational Rose does not offer the same flexibility as UMLet does with the modification of the type of a dependency. Within Rational Rose the user has either to delete and re-create the new dependency or do the modification (if even possible) within a dialog. For example if the user creates an aggregation of two classes. So he connects `Class1` with `Class2` and may find that the aggregation he made follows the wrong direction. Rational Rose requires the user to do the following:

1. Open the context menu of the aggregation.
2. Select "Open Specification..." to open the dialog.
3. Switch to the tab "Role B Detail" and uncheck the Aggregate-check box.
4. Switch to the tab "Role A Detail" and activate the corresponding check box.

UMLet allows one to do this type of modification with a fraction of activities and time. The user just has to modify the textual attributes of the aggregation and replace the string "`->>>>`" to "`<<<<-`" or swap the beginning and end of the aggregation in the diagram. So UMLet offers an intuitive way of changing the characteristics of dependencies and other UML elements. UMLet even lets the user change the type of dependency by modifying the attribute string by removing or adding a "`<`" or "`>`".

6.1.7 Summary

All these mentioned issues hamper the user when he just wants to sketch simple diagrams. Rational Rose is too complicated and overloaded for this purpose.

UMLet allows the user to get used to the tool in a much shorter time than Rational Rose does. UMLet allows the user to start sketching right after the application start. There are no questions or setup options. UMLet offers in its default palette the most common element types.

UMLet features a more agile approach than Rational Rose. UMLet allows fast sketching and refining the diagram on demand rather than forcing the user to specify any details.

6.2 Evaluation

The Unified Modeling Language (UML) [9] has become the standard graphical notation in software engineering. Different diagram types support most phases and workflows of the software process, including requirement engineering, design, implementation and deployment. UML is supported by a variety of tools trying to deliver on the elusive promise of computer-aided software engineering [16].

In practice, UML is applied in three main scenarios. First, UML is the notation of choice when creating early drafts of requirement specifications, and software or database designs. Use case diagrams and design sketches are often created from scratch and modified over several iterations. The diagrams do not have to adhere to the strict UML standards; they are used in an explorative way [38, page 125].

Second, in large-scale software engineering environments, software design and implementation are kept in sync using sophisticated round-trip engineering tools [32]. These are capable of generating code stubs from design blueprints, of generating diagrams from existing code, and of propagating changes from one artifact to others. The models and diagrams must conform to formal criteria; only in this way are the tools able to handle the relations between diagrams and code.

Finally, such round-trip engineering tools are also able to effortlessly generate UML documentation from large existing code bases. This is an easy way to provide clients with a seemingly vast amount of system documentation, which is often required by contract but seldom maintained consistently during a project's lifetime.

This paper looks at the first application scenario—UML sketching—, and more precisely at tools supporting the sketching process. We argue that UML is applied differently in that scenario, and that tools aimed at providing formal UML support and complex round-trip-engineering might be inadequate to cover it. We compare the commercial UML tool Rational Rose to the open-source tool UMLet [6] and quantitatively assess the tools' usability for explorative sketching. Rational Rose was chosen because it is the leading UML modeling tool in large-scale industrial environments. UMLet, on the other hand, provides a low-complexity user interface and thus a baseline for our comparison.

We assess the tools' usability by measuring the complexity of 16 common UML modeling patterns, or *use cases*. Examples for such use cases are changing class attributes, modifying dependencies, or adding messages in a UML diagram. The applied usability measures rely on concepts outlined in [36].

6.3 Hypothesis

Classical text-based applications like command line interfaces (i.e. Unix) are known to be complex and critical to erroneous input (i.e. using the `rm` command with wildcards). On the other side graphical user interfaces can become slow and cumbersome—looking for a specific icon within a large list of other abstract and hardly decipherable icons.

The unobtrusive combination of both—a graphical user interface combined with a smart text-based interface can improve the application's usability. Hence the research hypothesis is that ***text-based user input can reduce the number of user interactions with a graphical user interface, increase the speed of user input and lead to a more efficient user interface.***

This hypothesis is tested using a quantitative method counting user interactions needed to fulfill given use cases. The assessed tools are the industry-standard tool Rational Rose and the open-source tool UMLet. Rational Rose features a classical graphical user interface. UMLet's user interface design reduces the amount of distraction caused by typical UI elements like pop-up windows, tab-panes, abstract icons, etc. It features a text-based approach with an easy to learn syntax.

The dataset which the assessment is based on, features 16 use cases that are essential to UML sketching. See below for their detailed description.

6.4 Evaluated UML tools

This section gives a short introduction to the evaluated UML tools: one of the leading industry-strength UML tools, Rational Rose, and the open-source tool UMLet.

6.4.1 Rational Rose

Rational Rose is one of the leading UML tools used in large-scale software development environments. Rational Rose is no longer sold as a stand alone application, but it is integrated into a number of CASE software products offered by IBM.

Rational Rose is not intended to be used as a simple sketching tool—it aims at providing assistance in designing large software systems. It relies on a formal, internal object model that allows one to view the design from different model perspectives.

Rational Rose was originally developed in 1992 at Rational Software Labs, using ADA and later Smalltalk. Version 1.0 supported the Booch notation only, since it was based on a tool developed by Grady Booch that created graphical representations of ADA program structures. Version 2.0 was released in 1993, supporting Microsoft

Windows. After James Rumbaugh joined Rational, version 3.0 supported the OMT notation and featured some of the first round-trip engineering capability for C++. In 1996, Rose 4.0 was released, including Ivar Jacobson's use cases, improved round-trip engineering, and basic support for Visual Basic. Rose 98 featured UML notation, activity diagrams, and Java support. Rose 2000 shipped with an HTML generator and increased UML conformity. Rose 2001 supported J2EE and IBM's VisualAge for Java. Rational was acquired by IBM in 2003.

Model views and round-trip engineering

Rational Rose uses a formal model framework to store all design elements and their relationships. This way, Rational Rose can create different views of the design and ease the transformation of one diagram type to another. As an example, if the user changes the name of an element in one diagram view, it changes consistently in all other diagrams.

Rational Rose allows one to create a model from scratch or to start from a set of predefined models. Users can select from model templates like Java, VC-MFC, VB, etc. Figure 6.6 shows Rose's model browser. It is used to navigate through the model, and to add and delete diagrams, entities and relationships.

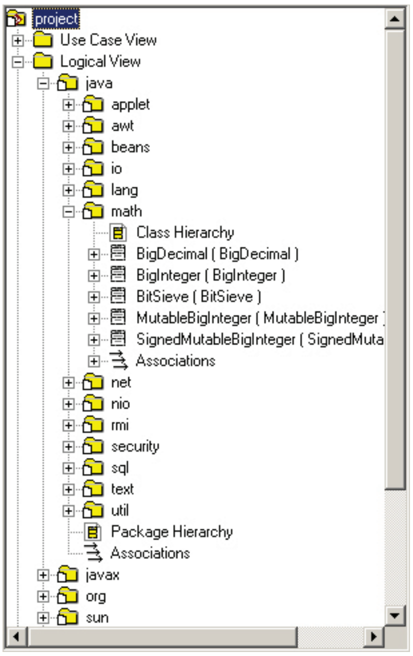


Figure 6.6: Rational Rose's model browser

A key feature of Rational Rose is its round-trip engineering capability which allows one to generate source code from the model and model elements from source code. Round trip engineering consists of two parts:

- *Forward Engineering*: Changes to the UML model are translated into source code changes.

- *Reverse Engineering*: Changes to the source code are updated in the model's elements.

In our own experience, and based on reports from several fellow IT managers, this round-trip engineering process is not flawless. Large-scale industrial IT projects involve several different programming languages, operating systems, databases, class frameworks and batch scripts, in addition to a version control system, a code documentation tool, etc. This diversity alone makes keeping these systems and artifacts in sync a complex challenge.

6.4.2 UMLet: Lightweight UML modeling

UMLet is a small UML sketching tool. It aims at early life-cycle UML modeling and UML education. It is distributed as open source tool under the terms of the GNU General Public License. Since it is developed in Java it is operating system independent. UMLet may also be used as a plug-in within the integrated development environment Eclipse to better integrate UML models with a project's source code artifacts.

UMLet was originally developed in 2001 at the Vienna University of Technology. The first versions were designed to run as an applet in a browser, with diagrams being stored on a central server. The setup of this client/server solution proved too tedious for average UML users and students; the following versions were therefore released as a stand-alone Java application. Several features were added in the following years: versions 1 to 3 provided export capabilities, integration in the Eclipse IDE, and user-defined element palettes. Versions 4 to 7 added new UML diagram types and user-defined UML elements via on-the-fly Java compilation. The main user interface concept remains the text-based UML element description.

Text-based modeling

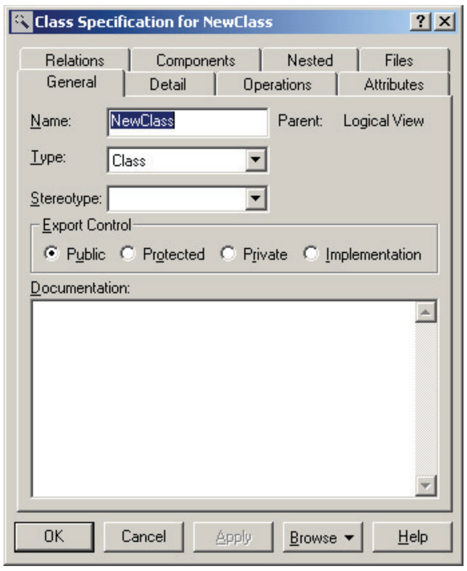
UML tools usually treat UML elements as visual objects, whose appearance can be edited by changing their attributes. This is mostly done through pop-up dialog boxes. See figure 6.7 for an example of Rose's dialog to edit a UML class element and its attributes. The dialog contains 8 tabs, and approximately 40 user interface elements.

UMLet's user interface is different: it allows users to define the look of a UML element by editing a textual description of it. For example, the UML class element in figure 6.8 is defined by the following lines:

```
MyClass
--
id: Long
ClassAttribute: Long
--
MyClass(i: int)
someOperation(): Object
```

The double dash denotes the lines that are separating class title, attributes and methods. Changing the title, or adding and removing new attributes and methods is done by editing the textual description of the class.

Not only simple UML elements like classes can be modified like this. UMLet also provides more complex diagram types entirely defined by a text grammar. The sequence diagram of figure 6.9, for example, is defined as follows:



Class Specification for NewClass

Name: NewClass
Parent: Logical View
Type: Class
Stereotype:
Export Control: [✓ Public] Protected Private Implementation
Documentation:

Figure 6.7: Rational Rose: Class Specification Dialog

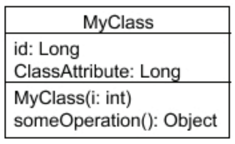


Figure 6.8: Class element

```
title: Sequence Diagram
_alpha:A_|_beta:B_|_gamma:G_
1->>2:1,2
2-/>1:async Msg.
3->>>1:1,3
1.>3:1,3:async return Msg
1->1:1:self
iframe{:interaction frame
2-/>3:2,3:async Msg.
iframe}
```

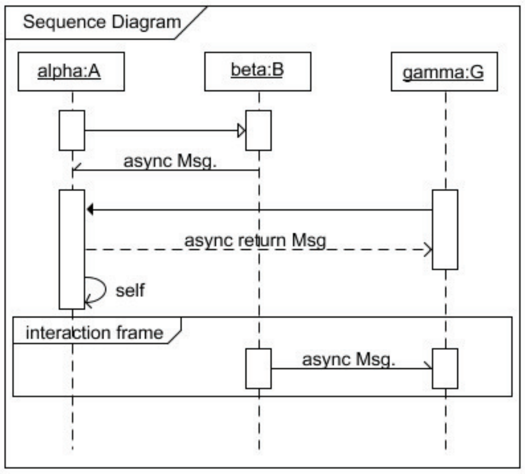
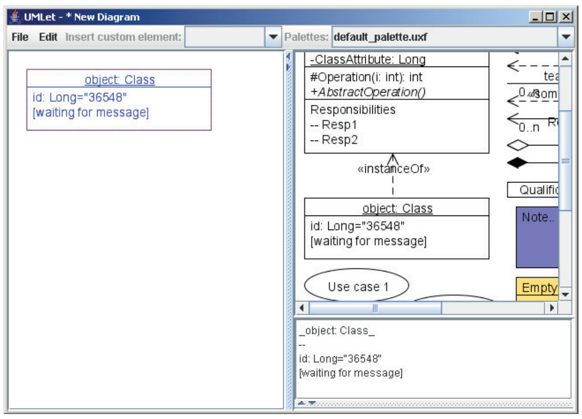


Figure 6.9: Sequence diagram using a simple grammar

When the textual description of a UML element is edited by the user, the element's graphical representation is updated in real time—no separate dialog or confirmation is necessary. The text editor also provides standard copy/paste functionality, which is useful, for example, when adding several private attributes and corresponding get/set-methods to a class.

Figure 6.10 shows the user interface's three main parts: diagram panel, element palette panel and the text panel to edit the UML element attributes.

UMLet's element palettes are normal UML diagrams: they show the available diagram elements in real size rather than as tiny, abstract icons. The users can quickly identify the elements without having to interpret icons or navigate menus; see figure 6.10. And as a palette is just another UML diagram, it can be rearranged or modified to show the most useful elements or element configurations for a given environment.



```
_object: Class_
--
id: Long="36548"
[waiting for message]
```

Figure 6.10: UMLet's user interface

6.5 UML tool usage

This section describes the three main usages of UML tools: explorative UML sketching, round-trip engineering, and system documentation. To perform the complex round-trip engineering and documentation, tools must enforce formal language constructs more rigorously. They can thus become unsuitable for simple explorative sketching.

Explorative UML modeling is the fast creation of UML sketches. Those sketches often do not have to be precise or final or completely UML compliant. The goal of explorative sketching is to play and experiment with the model, to reach a better idea of the design iteratively, and to discuss the model with colleagues and other stakeholders in the requirements analysis phase. Explorative sketching may also be used in UML education—UML examples can focus on key language elements while disregarding distracting details. A simple and fast modification of the diagram and its elements is crucial.

Round-trip engineering aims to keep the design and the actual implementation of the software project synchronized. This should guarantee that the high-level abstraction and the low-level both implementation remain valid over the project's lifetime. This is important, for example, if changes to an application's overall software architecture should be made—the abstract design view is the natural starting point for such a refactoring effort. If the design view, however, does no longer represent the actual implementation, the refactoring must start on the implementation level and it becomes less transparent and more tedious.

Documenting existing artifacts is another major application for UML tools. It is often motivated by the fact that large projects require extensive system documentation. Unfortunately, a common measure for the quality of the documentation is its size, as precision and understandability are difficult to assess. An easy way to create massive amounts of documentation is to reverse-engineer diagrams from existing code. These generated diagrams are often very hard to interpret since, unlike hand made diagrams, they can't hide unnecessary details or make use of the elements' spatial orientation.

This paper focuses on the first application of UML tools, UML sketching. This is a fundamental UML tool application—every design tool, at several points in a project's life cycle, is likely to be used as a UML sketchpad. If it is too cumbersome to use, users will turn away and settle for improvised PowerPoint graphs or scanned notes.

6.6 Testing scenarios and rules

There are several methods to examine the usability and ergonomics of user interfaces. The most straightforward method might be to examine user behavior while they execute well defined use cases. This method can be combined with video recordings or even eye tracking recordings. Examinations like this, however, are very time consuming and expensive.

Raskin [36] discusses an alternative method for quantifying user interface usability: GOMS (Goals, Operators, Methods, and Selections rules). GOMS, developed by Stuart Card, Thomas Moran and Allen Newell, evaluates a user interface by analyzing elementary actions like pointing and clicking with the mouse, or typing on the keyboard. These elementary actions are weighted with time factors.

We simplify this method by concentrating on mouse clicks and combined keyboard inputs, while disregarding mouse movements or individual key presses. We also do not weight the elementary actions, and only count the number of actions needed to complete a given use case.

The following list gives a set of representative use cases which are typical to the creation of UML diagrams. We concentrate on class diagrams and sequence diagrams, since the creation and modification of other UML diagram types is very similar.

1. **Create a simple class:** Starting from an empty diagram we create a simple class element without defining any attributes or operations.
2. **Extend a simple class with attributes:** Extend the simple class with one attribute without specifying special characteristics.
3. **Extend a simple class with operations:** Extend the (simple) class with one operation without specifying a special return value or input parameters.
4. **Modify an attribute's characteristics:** Modify the attribute; define it to be protected and specify its type as *Object*.
5. **Duplicate a class:** Make a copy of the class.
6. **Add an aggregation to a two-class diagram:** Add an aggregation dependency between two classes.
7. **Modify an aggregation to a generalization:** Change the aggregation dependency to a generalization dependency.
8. **Change the direction of a generalization:** Change the direction of the generalization, so the specialized class becomes the parent class.
9. **Delete one class:** Remove one class from the diagram.
10. **Undo class delete:** Undo removing one class from the diagram.

11. **Create a simple class diagram:** Create a slightly more complex class diagram. The composite design pattern—consisting of 4 classes and three different relationship types—is implemented.
12. **Create a simple sequence diagram:** Create a simple sequence diagram consisting of two objects. Object *One* sends a synchronous message to object *Two*.
13. **Change the message direction:** Change the direction of a message in a sequence diagram.
14. **Change the message type:** Change the message type from synchronous to asynchronous.
15. **Add a message to the sequence diagram:** Add a named message to the sequence diagram from object *One* to object *Two*.
16. **Create a sequence diagram:** Create a sequence diagram of the process of browsing with a Web browser to a Web site. The involved objects are *User*, *Browser*, *Web server*, *DNS*. Object types and message types are not specified.

As an example, these are the individual user interactions required for use case 1, with Rational Rose and UMLet:

Create a simple class:

Starting from an empty diagram we create a simple class element without defining any attributes or operations.

- Starting point: An empty diagram
- Goal: A diagram with a simple class element

User interactions required by UMLet: 2

1. Create the class by double clicking on the simple class on the palette.
2. Rename the class to "`MyClass`".

User interactions required by Rational Rose: 4

1. Select the class element by clicking on the class icon on the tool bar.
2. Place the class element by clicking in the diagram window.
3. Rename the class to "`MyClass`".
4. Click into the diagram window to complete the naming operation.

For an extensive description of all use cases, please refer to the appendix.

6.7 Discussion

The results of the test are summarized in table 6.1 and figure 6.11. It shows that UMLet requires substantially fewer user interactions than Rational Rose; there are only a few tasks that can be executed faster in Rational Rose. To complete simple but frequent tasks when creating UML sketches, users are required to

#	Use Case	# UI UMLet	# UI R.Rose	Abs. Diff.	Rel. Diff.
1	Create a class	2	4	+2	100%
2	Extend class with attributes	5	4	-1	-20%
3	Extend class with one operation	5	4	-1	-20%
4	Modify an attribute	5	10	+5	100%
5	Duplicate a class	1	11	+10	1000%
6	Add an aggregation	4	2	-2	-50%
7	Modify an aggregation	1	5	+4	400%
8	Change a generalization	1	5	+4	400%
9	Delete a class	2	3	+1	50%
10	Undo class delete	2	2	+/-0	0%
11	Simple class diagram	39	48	+9	23%
12	Simple sequence diagram	4	29	+25	625%
13	Change message direction	3	11	+8	266%
14	Change message type	3	5	+2	66%
15	Add a message	3	4	+1	33%
16	Create a sequence diagram	12	49	+37	308%
				Median Relative Difference	83%

# UI = number of user interactions

Table 6.1: Results

perform about 80% more interactions with Rational Rose than UMLet.[1]

Surprisingly, duplicating a class element (use case 5) in Rational Rose is an extremely cost intensive task. It requires 11 interactions in Rose, and just one in UMLet.

Apart from use case 5, use case 16 shows the maximum performance advantage of all tested use cases. The reason is UMLet's special grammar for sequence diagrams. It especially frees the user from the task of placing and resizing the various sub-elements' graphical representations. In order to keep the grammar's syntax simple, UMLet makes some trade-offs and does not support the full functional range of the UML sequence diagram. UMLet does provide support for creating such diagrams conventionally, using individually placed elements. In this case, however, UMLet's advantage shrinks.

As documented in table 6.1, operations like changing the direction of dependencies (use case 8) or changing the type of dependencies (use case 7) are very simple using UMLet. Rational Rose's way of dealing with this is more time-consuming. In theory, this seems be o.k.—after all, a dependency's direction or type seems to be so fundamental that changing it does not make sense in most cases. We found, however, that these use cases actually occur quite often when sketching diagrams, especially if the wrong relation was inadvertently added to the diagram, if the classes responsibilities change, or if inheritance structures are broken up and changed to looser class relations.

Although not covered by our test, even the simple task of adding multiplicities can turn out to be a struggle. Rational Rose offers two ways of adding multiplicities to a relation. Either the user selects it from a list box in one of the tabs of the specification dialog (requiring 6 user interactions), or the user selects the multiplicity type from the context menu (requiring 3 user interactions). The

[1]The sign test rejects the null hypothesis of a zero median difference; it is significant at the 5% level.

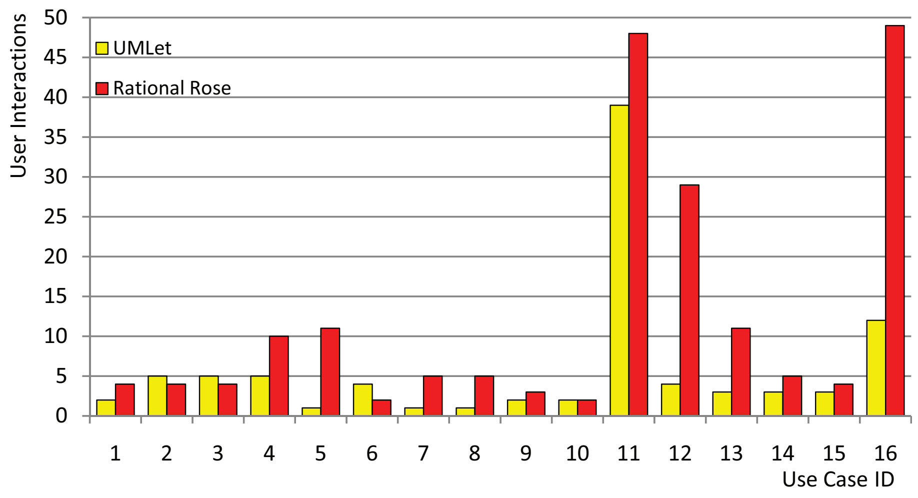


Figure 6.11: Results

problem is that the user then often realizes that the multiplicity was added at the wrong side of the relation. So he is often forced to delete the multiplicity and re-add it. Mistakes like these occur in UMLet, too, but the correction can be done in a single user interaction using the text-based attribute specification, whereas Rose requires the entire process to be repeated.

The comparison of the two tools should not determine a "better" UML tool—after all, the tools have widely different aims. UMLet, in this paper, should merely denote a baseline, a low-complexity approach to which Rational Rose, one of the leading UML tools, can be compared with respect to fast UML sketching. Our comparison indicates that Rational Rose, on average, requires almost two times as many user interactions as necessary.

Is this merely the consequence of the fact that Rational Rose has many more features, and has to enforce more formal UML standards? Or, more generally, do tools dealing with complex demands necessarily become more complex?

We don't think so. A good example is Microsoft's suite of integrated development environments, Visual Studio. While offering an astonishing array of options and features, the basic functionalities of programming—typing, searching, looking up object members—continue to be very easy to use. The search functionality was actually improved over time (it is now non-modal, and offers a history of searches) without sacrificing usability; the object member lookup still hides complex functionality behind an unobtrusive and efficient user interface.

Tools thus are able to both tackle complex requirements, and to provide intuitive base functionality. In the quest to offer ever-more features, this goal should not be neglected.

6.8 Conclusion and further research

This paper compares two UML tools with respect to their suitability for explorative UML sketching. Several common UML design tasks were tested to determine the number of required user interactions.

The large, standard-conforming and -enforcing Rational Rose was found to require substantially more user interactions than UMLet. As Rational Rose's design goals have to accommodate a wide range of requirements, fast and explorative UML sketching becomes less intuitive and more tedious. This is assessed by comparing Rational Rose to UMLet, a tool specifically tailored to creating UML sketches.

We argue that as tools get more complex, developers must make sure to avoid compromising on important base functionality—otherwise, a tool will cover more requirements, but important ones less well.

Further research will focus on

- aspects of tool complexity and integration, especially on ways to integrate separate interactive and highly graphical applications;

- refined user interaction measures, that take into account not just the number of user interactions, but their type and complexity (like decoding an icon's meaning, or clicking on small, scattered buttons).

6.9 Appendix: Use case description

6.9.1 Create a simple class

Starting from an empty diagram we create a simple class element without defining any attributes or operations.

- Starting point: An empty diagram

- Goal: A diagram with a simple class element

User interactions required by UMLet: 2

1. Create the class by double clicking on the simple class on the palette.
2. Rename the class to "`MyClass`".

User interactions required by Rational Rose: 4

1. Select the class element by clicking on the class icon on the tool bar.
2. Place the class element by clicking in the diagram window.
3. Rename the class to "`MyClass`".
4. Click into the diagram window to complete the naming operation.

6.9.2 Extend a simple class with attributes

Extend the simple class with one attribute without specifying special characteristics.

- Starting point: A simple class element
- Goal: A class with one attribute

User interactions required by UMLet: 5

1. Select the class element by clicking on it.
2. Click into the text attributes window.
3. Specify the separation line by adding the "`--`"-line to the textual representation.
4. Add the attribute's name.
5. Resize the class to make the attribute fully visible.

User interactions required by Rational Rose: 4

1. Open the context menu of the class element by right clicking on it.
2. Select "`New Attribute`" from the context menu.
3. Type the attribute's name.
4. Click into the diagram window to complete the naming operation.

6.9.3 Extend a simple class with operations

Extend the (simple) class with one operation without specifying a special return value or input parameters.

- Starting point: A (simple) class element
- Goal: A class with one operation

User interactions required by UMLet: 5

1. Select the class element by clicking on it.
2. Click into the text attributes window.
3. Specify the separation line by adding the "`--`"-line to the textual representation.
4. Add the operation's name.
5. Resize the class to make the operation fully visible.

User interactions required by Rational Rose: 4

1. Open the context menu of the class element by right clicking on it.
2. Select "`New Operation`" from the context menu.
3. Type the operation's name.
4. Click into the diagram window to complete the naming operation.

6.9.4 Modify an attribute's characteristics

Modify the attribute; define it to be protected and specify its type to *Object*.

- Starting point: A class element with an attribute
- Goal: A class with a more precisely specified attribute

User interactions required by UMLet: 5

1. Select the class element by clicking on it.
2. Click into the text attributes window.
3. Add the protected flag "`#`" in front of the attributes name.
4. Add the attribute's type (Object) after the attribute's name.
5. Resize the class to make the attribute and its characteristics fully visible.

User interactions required by Rational Rose: 10

1. Open the context menu of the class element by right clicking on it.
2. Select "`Open Specification`" to open the *Class Specification* dialog.
3. Select the tab "`Attributes`".
4. Double click on the attribute to open the *Class Attribute Specification* dialog.
5. Open the pull-down list selector from the "`Type`" box.
6. Search "Object" in the list by scrolling through the list.
7. Select "Object" from the list by clicking on the item.
8. Choose the "Protected" radio button from the "Export Control" frame.
9. Close the *Class Attribute Specification* dialog by clicking *OK*.
10. Close the *Class Specification* dialog by clicking the *OK* button.

6.9.5 Duplicate a class

Make a copy of the class.

- Starting point: One class element

- Goal: Two class elements

User interactions required by UMLet: 1

1. Double click on the class in the diagram.

User interactions required by Rational Rose: 11

1. Open the context menu of the class diagram by right clicking on free space in the diagram.
2. Select "Class wizard..." from the context menu.
3. Select the "Clone the class based on an existing class" radio button.
4. Press the *Next* button.
5. Select the class to be duplicated from the list box.
6. Press the *Next* button.
7. Press the *Next* button on the *Class Documentation* screen.
8. Press the *Next* button on the *Parent Category* screen.
9. Select the class diagram into which the clone shall be placed from the list box.
10. Press the *Next* button.
11. Press the *Finish* button.

6.9.6 Add an aggregation to a two-class diagram

Add an aggregation dependency between two classes.

- Starting point: A diagram with two classes
- Goal: Two classes with an aggregation dependency

User interactions required by UMLet: 4

1. Scroll the palette panel to the right to see the offered arrow types completely.
2. Create an instance of the aggregation arrow by double clicking on it.
3. Move the head of the aggregation and connect it to the first class.
4. Move the tail of the aggregation and connect it to the second class.

User interactions required by Rational Rose: 2

1. Select the *Aggregation* tool from the tool bar.
2. Create the aggregation by pressing the left mouse button over the first class, dragging over to the second class, and releasing the button.

6.9.7 Modify an aggregation to a generalization

Change the aggregation to a generalization.

- Starting point: Two classes with an aggregation dependency
- Goal: Two classes with a generalization dependency

User interactions required by UMLet: 1

1. Change the definition string of the aggregation from "`<<<<-`" to "`<<-`" in the attributes panel.

User interactions required by Rational Rose: 5

1. Open the context menu of the aggregation by right clicking on it.
2. Select the sub-menu *Edit* from the context menu.
3. Select *Delete from Model* from the sub-menu.
4. Select the *Generalization* tool from the tool bar.
5. Create the generalization by pressing the left mouse button over the first class (Class2), dragging over to the second class (Class1), and releasing the button.

6.9.8 Change the direction of a generalization

Change the direction of the generalization.

- Starting point: Two classes with a generalization dependency: Class2 specializes Class1.
- Goal: Two classes with a generalization dependency: Class1 specializes Class2.

User interactions required by UMLet: 1

1. Change the definition string of the generalization from "`<<-`" to "`->>`" in the attributes panel.

User interactions required by Rational Rose: 5

1. Open the context menu of the aggregation by right clicking on it.
2. Select the sub-menu *Edit* from the context menu.
3. Select *Delete from Model* from the sub-menu.
4. Select the *Generalization* tool from the tool bar.
5. Create the generalization by pressing the left mouse button over the first class (Class1), dragging over to the second class (Class2), and releasing the button.

6.9.9 Delete one class

Remove one class from the diagram.

- Starting point: A diagram with two classes
- Goal: A diagram with one class

User interactions required by UMLet: 2

1. Open the context menu of the to be deleted element by right clicking it.
2. Select the menu item *Delete*.

User interactions required by Rational Rose: 3

1. Open the context menu of the aggregation by right clicking on it.
2. Select the sub-menu *Edit* from the context menu.
3. Select *Delete from Model* from the sub-menu.

6.9.10 Undo class delete

Undo removing one class from the diagram.

- Starting point: A diagram with one class after deleting a second class
- Goal: A diagram with two classes according to the diagram before deleting the class

User interactions required by UMLet: 2

1. Select the *Edit* menu.
2. Select the *Undo* menu item.

User interactions required by Rational Rose: 2

1. Select the *Edit* menu.
2. Select the *Undo Delete* menu item.

6.9.11 Create a simple class diagram

Create a slightly more complex class diagram. The *composite* design pattern consists of 4 classes and three different relationship types. Figure 6.12 and figure 6.13 show the results of UMLet and Rational Rose.

- Starting point: An empty class diagram.
- Goal: A diagram of the composite design pattern.

User interactions required by UMLet: 39

1. Create a class by double clicking on the simple class at the palette.
2. Click into the text attributes window.
3. Rename the class to “`Client`”.
4. Create a class by double clicking on the simple class at the palette.
5. Move the class.
6. Click into the text attributes window.
7. Rename the class to “`Component`”.
8. Add operation `operation()`.
9. Add operation `add(Component)`.

10. Add operation `remove(Component)`.
11. Add operation `get(index)`.
12. Resize class element.
13. Create a class by double clicking on the simple class at the palette.
14. Move the class.
15. Click into the text attributes window.
16. Rename the class to "`Leaf`".
17. Add operation operation().
18. Resize class element.
19. Duplicate class *Component* by double clicking on it.
20. Move the class element.
21. Click into the text attributes window.
22. Rename the class to "`Composite`".
23. Create arrow of type association by double clicking on it in the palette.
24. Move arrow tail to *Client*.
25. move arrow head to *Component*.
26. Create arrow of type generalization by double clicking on it in the palette.
27. Move arrow tail to Leaf.
28. Move arrow head to Component.
29. Duplicate generalization arrow by double clicking on it.
30. Move arrow tail to Composite.
31. Move arrow heat to Component.
32. Create arrow of type aggregation by double clicking on it in the palette.
33. Move tail of the arrow to Composite.
34. Move head of the arrow to Component.
35. Create support point by clicking on the aggregation and moving the mouse.
36. Create support point by clicking on the aggregation and moving the mouse.
37. Click into the text attributes window.
38. Add "`>`" to the textual description of the aggregation to make it directional.
39. Add multiplicity by typing "`m2=*`".

User interactions required by Rational Rose: 48

1. Select the class creation tool from the tool bar.
2. Click into the diagram to place the class element.
3. Name the class `Client`.
4. Click outside of the class to end the class creation process.
5. Select the class creation tool from the tool bar.
6. Place the class element.
7. Name the class `Component`.
8. Right click on *Component* to open its context menu.
9. Select "New Operation".
10. Type name of the new operation `operation`.
11. Type name of the new operation `add`.

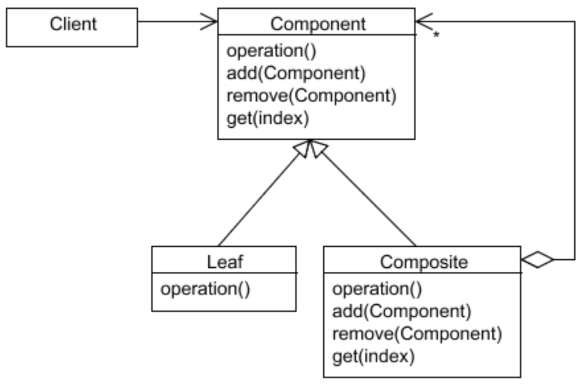


Figure 6.12: Composite pattern using UMLet

12. Type name of the new operation `remove`.
13. Type name of the new operation `get`.
14. Click outside of the class to end the operation creation process.
15. Select the class creation tool from the tool bar.
16. Place the class element.
17. Name the class `Leaf`.
18. Right click on *Leaf* to open its context menu.
19. Select "New Operation".
20. Type name of the new operation `operation`.
21. Click outside of the class to end the operation creation process.
22. Right click on free space in the diagram to open the context menu.
23. Select "Class Wizard".
24. Select "Clone the class based on an existing class".
25. Click the "Next" button.
26. Select *Component* from the selection box.
27. Click the "Next" button.
28. Select the text of the new class name.
29. Change the name to `Composite`.
30. Move *Composite*.
31. Click the "Next" button.
32. Click the "Next" button.
33. Click the "Next" button.
34. Click the "Finish" button.
35. Select "Unidirectional Association" creation tool from the tool bar.
36. Click on *Client* and drag mouse to *Component*.
37. Move *Client*.

38. Select “Generalization” tool from the tool bar.
39. Click on *Leaf* and drag the mouse to *Component*.
40. Select “Generalization” tool from the tool bar.
41. Click on *Composite* and drag the mouse to *Component*.
42. Select “Unidirectional Aggregation tool from the tool bar.
43. Click on *Composite* and drag the mouse to *Component*.
44. Create support point by clicking on the aggregation and moving the mouse.
45. Create support point by clicking on the aggregation and moving the mouse.
46. Right click on the aggregation to open its context menu.
47. Select the submenu “Multiplicity”.
48. Select “n”.

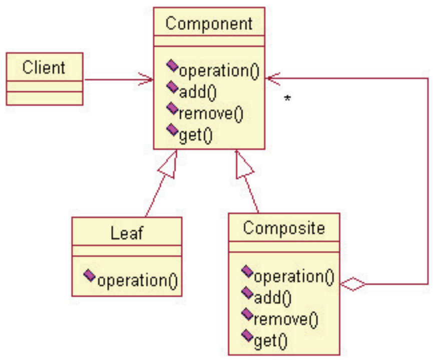


Figure 6.13: Composite pattern using Rational Rose

6.9.12 Create a simple sequence diagram

Create a simple sequence diagram consisting of two objects. Object One sends a synchronous message to object Two. Figure 6.14 and figure 6.15 show the created diagrams.

- Starting point: An empty diagram.
- Goal: A sequence diagram with 2 objects and a synchronous message from object One to object Two.

User interactions required by UMLet: 4

1. Create a new sequence diagram by double clicking the template diagram from the sequence palette.
2. Select the text from the textual description and delete it.
3. Create both object of the sequence diagram by typing “`_One:Object_|_Two:Object_`”.
4. Create the synchronous message by typing “`1->>>2:1,2:message`”

User interactions required by Rational Rose: 29

1. Right click on “Logical View” or “Use Case View” in the model browser to open the context menu.
2. Select the “New...” sub menu.
3. Select “Sequence Diagram”.

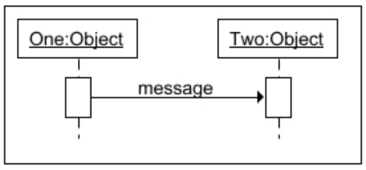


Figure 6.14: UMLet: Simple sequence diagram

4. Name the diagram.
5. Open the diagram window by double clicking on it.
6. Select the “Object” creation tool from the tool bar.
7. Place the object in the diagram.
8. Name the object `One`.
9. Right click on *One* to open its context menu.
10. Select “Open Specification...”.
11. Click on the class selector to open the list box.
12. Type `Object` into the “Name” field of the *Class Specification* dialog.
13. Click the “OK” button to close the *Class Specification* dialog.
14. Click the “OK” button to close the *Object Specification* dialog.
15. Select the “Object” creation tool from the tool bar.
16. Place the object in the diagram.
17. Name the object `Two`.
18. Right click on *Two* to open its context menu.
19. Click on the class selector to open the list box.
20. Select “Object”.
21. Click on the “OK” button to close the *Object Specification* dialog.
22. Select “Object Message” from the tool bar.
23. Click into the life mark of *One* and drag the mouse to the life mark of *Two*.
24. Right click on the message to open its context menu.
25. Select “Open Specification...”.
26. Type “`message`” into the Name field.
27. Select the “Detail” tab.
28. Choose the “Synchronous” radio button.
29. Click the “OK” button.

6.9.13 Change the message direction

Change the direction of a message in a sequence diagram.

- Starting point: A sequence diagram with 2 objects and a synchronous message from object One to object Two.
- Goal: A sequence diagram with 2 objects and a synchronous message from object Two to object One.

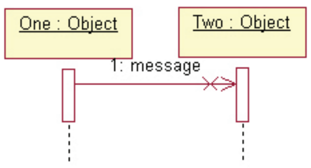


Figure 6.15: Rational Rose: Simple sequence diagram

User interactions required by UMLet: 3

1. Select the sequence diagram in the diagram panel.
2. Select the text panel.
3. Change the textual representation of the message type from "`->>>`" to "`<<<-`".

User interactions required by Rational Rose: 11

1. Right click on the message to open its context menu.
2. Select the sub-menu *Edit* from the context menu.
3. Select *Delete from Model* from the sub-menu.
4. Select "Object Message" from the tool bar.
5. Click into the life mark of *Two* and drag the mouse to the life mark of *One*.
6. Right click on the message to open its context menu.
7. Select "Open Specification...".
8. Type "`message`" into the Name field.
9. Select the "Detail" tab.
10. Choose the "Synchronous" radio button.
11. Click the "OK" button.

6.9.14 Change the message flavor

Change the message type from synchronous to asynchronous.

- Starting point: A sequence diagram with 2 objects and a synchronous message from object Two to object One.
- Goal: A sequence diagram with 2 objects and an asynchronous message from object Two to object One.

User interactions required by UMLet: 3

1. Select the sequence diagram in the diagram panel.
2. Select the text panel.
3. Change the textual representation of the message type from "`<<<-`" to "`</-`".

User interactions required by Rational Rose: 5

1. Right click on the message to open its context menu.
2. Select "Open Specification...".
3. Select the "Detail" tab.
4. Choose the "Synchronous" radio button.
5. Click the "OK" button.

6.9.15 Add a message to the sequence diagram

Add a named message to the sequence diagram from object One to object Two. The message type is not specified.

- Starting point: A sequence diagram with 2 objects and one message with a name.
- Goal: A sequence diagram with 2 objects and two messages with names.

User interactions required by UMLet: 3

1. Select the sequence diagram in the diagram panel.
2. Select the text panel.
3. Add the message by typing: “`1->2:1,2:message2`”.

User interactions required by Rational Rose: 4

1. Select “Object Message” from the tool bar.
2. Click into the life mark of *One* and drag the mouse to the life mark of *Two*.
3. Type “`message2`” to name the message.
4. End the input process by clicking outside somewhere in the diagram, or by pressing Enter.

6.9.16 Create a sequence diagram

Create a sequence diagram of the process to browse with a Web browser to a Web site. The involved objects are *User*, *Browser*, *Web server*, *DNS*. Object types and message types are not specified.

- Starting point: An empty diagram.
- Goal: A sequence diagram with 4 objects and 9 messages.

User interactions required by UMLet: 12

1. Create a new sequence diagram by double clicking the template diagram from the sequence palette.
2. Select the text from the textual description and delete it.
3. Create all four objects of the sequence diagram by typing “`␣User␣|␣Browser␣|␣Webserver␣|␣DNS␣`”.
4. Create a synchronous message by typing “`1->>>2:1,2:Browse:  umlet.com`”
5. Create a synchronous message by typing “`2->>>4:1,2,4:DNS query:  url = umlet.com`”
6. Create a synchronous message by typing “`4.>>>2:1,2,4:DNS response:  IP-Address = 216.40.33.117`”
7. Create a synchronous message by typing “`2->>>3:1,2,3:SYN`”
8. Create a synchronous message by typing “`3->>>2:1,2,3:SYN+ACK`”
9. Create a synchronous message by typing “`2->>>3:1,2,3:ACK`”
10. Create a synchronous message by typing “`2->>>3:1,2,3:GET / HTTP`”
11. Create a synchronous message by typing “`3->>>2:1,2,3:HTTP 200 OK`”
12. Create a synchronous message by typing “`2.>>>1:1,2:Display`”

User interactions required by Rational Rose: 49

1. Right click on “Logical View” or “Use Case View” in the model browser to open the context menu.
2. Select the “New...” sub menu.
3. Select “Sequence Diagram”.
4. Name the diagram.

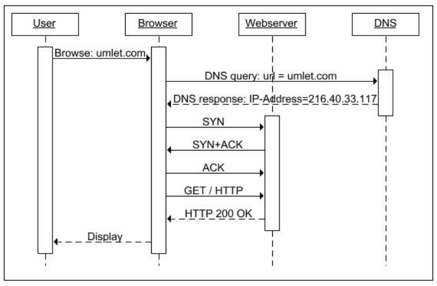


Figure 6.16: UMLet: Sequence diagram of use case 16

5. Open the diagram window by double clicking on it.
6. Select the “Object” creation tool from the tool bar.
7. Place the object in the diagram.
8. Name the object `User`.
9. Select the “Object” creation tool from the tool bar.
10. Place the object in the diagram.
11. Name the object `Browser`.
12. Select the “Object” creation tool from the tool bar.
13. Place the object in the diagram.
14. Name the object `Webserver`.
15. Select the “Object” creation tool from the tool bar.
16. Place the object in the diagram.
17. Name the object `DNS`.
18. Move object *Browser* (alignment)
19. Move object *Webserver* (alignment)
20. Move object *DNS* (alignment)
21. Select “Object Message” from the tool bar.
22. Click into the life mark of *User* and drag the mouse to the life mark of *Browser*.
23. Name the message: “`Browse:  umlet.com`”.
24. Select “Object Message” from the tool bar.
25. Click into the life mark of *Browser* and drag the mouse to the life mark of *DNS*.
26. Name the message: “`DNS query:  url = umlet.com`”.
27. Select “Return Message” from the tool bar.
28. Click into the life mark of *DNS* and drag the mouse to the life mark of *Browser*.
29. Name the message: “`DNS response:  IP-Addres = 216.40.33.117`”.
30. Select “Object Message” from the tool bar.
31. Click into the life mark of *Browser* and drag the mouse to the life mark of *Webserver*.
32. Name the message: “`SYN`”.
33. Select “Object Message” from the tool bar.

34. Click into the life mark of *Webserver* and drag the mouse to the life mark of *Browser*.
35. Name the message: “`SYN+ACK`”.
36. Select “Object Message” from the tool bar.
37. Click into the life mark of *Browser* and drag the mouse to the life mark of *Webserver*.
38. Name the message: “`ACK`”.
39. Select “Object Message” from the tool bar.
40. Click into the life mark of *Browser* and drag the mouse to the life mark of *Webserver*.
41. Name the message: “`GET / HTTP`”.
42. Select “Object Message” from the tool bar.
43. Click into the life mark of *Webserver* and drag the mouse to the life mark of *Browser*.
44. Name the message: “`HTTP 200 OK`”.
45. Click on the life mark of object *User*.
46. Resize the life mark to overhang the life mark of *Browser*.
47. Select “Return Message” from the tool bar.
48. Click into the life mark of *Browser* and drag the mouse to the life mark of *User*.
49. Name the message: “`Display`”.

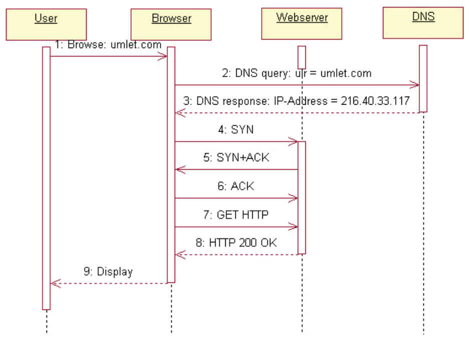


Figure 6.17: Rational Rose: Sequence diagram of use case 16

Chapter 7

Discussion

7.1 Summary

Lightweight tools aim at simple usage by reducing complexity to a minimum and reducing functionality to the basic needs. The UML offers a graphical notation inviting to sketch and play with the model in early design stages. Thus lightweight UML tools must cover the following requirements

1. Easy to use: A well designed user interface is essential for all software applications made for direct communication with humans.
2. Easy to learn: The design of the software and its workflows should be easily understandable and usable.
3. Easy to deploy: When deploying a software tool to more than one workstation and operating system, the process of software installation may be critical. Using a software tool for educational processes requires simple installation processes, especially when students are required to install the software on their home computers.
4. Relaxed standards restrictions: Explorative sketching is the process of creating not necessarily exact and complete diagrams. Tools that enforce strict standards conformity slow down the design process and draw too much attention to the software tool itself, than to the creation and development progress.

To achieve a simple and lightweight tool, UMLet's design focuses on the following concepts:

1. Simple standard file format: A simple XML format is human readable and can be transformed for the import into other tools.
2. Independence from platforms and operating systems using a 100% Java implementation.

3. Duality: Dual usage as a standalone application or as a plug-in for the Eclipse-IDE.

4. Transparency: Transparent implementation targeting the display device—screen, printer, file.

5. Easy expandability by providing custom palettes and dynamic custom elements featuring a built-in Java compiler and an in detail commented Java template.

6. Unobtrusiveness: The user interface avoids unnecessary distraction. It provides a shematized way of creating and modifying diagrams and elements, not compelling the user to handle differing data input varieties.

7. Fast diagram creation: UMLet supports the creation of simple elements and complex diagrams by its text-based input featuring simple grammar.

8. Interoperability: Support for a variety of standard export formats to interoperate with other tools.

9. Educational use: The easy integration into heterogeneous computer system environments, the simple distribution and installation process, the expandability on graphical notations, and the lightweight approach makes UMLet a neat tool in educational use issues.

Users may use and modify UMLet without any restrictions—it is published under the GPL and distributed as open source. It is designed to allow users to extend its functionality for their needs not only by being written using Java as the programming language. The emphasized use of design patterns helps users that want to extend UMLet understand how it is structured and how the components work together. Design patterns help one to understand the existing design as fast as possible.

The three main usages of UML tools: explorative UML sketching, round-trip engineering, and system documentation. Explorative UML modeling is the fast creation of UML sketches, which often do not have to be precise or final or completely UML compliant. Round-trip engineering focuses on the creation of code from models and vice-versa. Documenting existing artifacts is often motivated by the fact that large projects require extensive system documentation. Unfortunately, a common measure for the quality of the documentation is its size, as precision and understandability are difficult to assess. To perform the complex round-trip engineering and documentation, tools must enforce formal language constructs more rigorously. They can thus become unsuitable for simple explorative sketching.

7.2 Results

This section discusses how to compare the quality and effectiveness of user interfaces quantitatively. The applied generic method assesses tools using a set of relevant use cases that are essential to UML modeling.

Usability engineering and user experience design deal with making good user interfaces. Since it is hard to measure good design, there exists a number of guidelines (like Apple [2]), heuristics[1], best-practices and rules.
Quantifying and comparing the quality of user interfaces is a challenging task. Users with different experience levels judge differently on the evaluated interfaces. Raskin [36] provides quantifiable metrics to assess the quality of user interfaces. The GOMS methods measure user interfaces relying on 6 primitive operations (pressing a key, moving the mouse, dragging the mouse, mental preparation, moving hands, and waiting for command execution). The evaluation of two or more tools using a set of use cases relys on a simplified approach. Each sub-task of the use case is counted. Thus the evaluation can quantify the quality and efficiency of the user interfaces. Raskin states that user interfaces should be as monotonous as possible and modeless.

Chen and Zhang [10] tested two versions of an application—one featuring a text-based user interface, the other a graphical user interface. They found that experienced users preferred the text-based approach since it's more efficient interface, while novice users tended to use the graphical user interface. Their evaluation was based on the GOMS method as well.

Since Rational Rose is the de-facto industry standard of UML tools, all other serious competitors have to match with it. Both tools are assessed by applying a quantitative measuring on the user interface's complexity. The evaluation features 16 common use cases on UML modeling like changing class attributes, modifying dependencies, or adding messages:

1. Create a simple class
2. Extend a simple class with attributes
3. Extend a simple class with operations
4. Modify an attribute's characteristics
5. Duplicate a class
6. Add an aggregation to a two-class diagram
7. Modify an aggregation to a generalization
8. Change the direction of a generalization

[1]Jakob Nielsen: http://www.useit.com/papers/heuristic/heuristic_list.html

9. Delete one class
10. Undo class delete
11. Create a simple class diagram
12. Create a simple sequence diagram
13. Change the message direction
14. Change the message type
15. Add a message to the sequence diagram
16. Create a sequence diagram

As an example, these are the individual user interactions required for use case 1, with Rational Rose and UMLet:

Create a simple class:

Starting from an empty diagram we create a simple class element without defining any attributes or operations.

- Starting point: An empty diagram
- Goal: A diagram with a simple class element

User interactions required by UMLet: 2

1. Create the class by double clicking on the simple class on the palette.
2. Rename the class to "`MyClass`".

User interactions required by Rational Rose: 4

1. Select the class element by clicking on the class icon on the tool bar.
2. Place the class element by clicking in the diagram window.
3. Rename the class to "`MyClass`".
4. Click into the diagram window to complete the naming operation.

The results of the evaluation of all 16 use-cases on both tools shows that users of Rational Rose are required to perform about 80% more interactions. This is a result of UMLet's design goals of *unobtrusiveness* by avoiding unnecessary pop-ups and providing a simple and monotonous user interface, and *fast diagram creation* featuring text-based element creation. Use case 16 demonstrates UMLet's power of grammar-based diagram creation. The creation of sequence diagrams by textual descriptions is extremly efficient in comparison to conventional user interface approaches.

These findings are not only applicable on UML tools, but to a wide range of applications. So far text-based input often was limited to configuration files

#	**Use Case**	**# UI UMLet**	**# UI R.Rose**	**Abs. Diff.**	**Rel. Diff.**
1	Create a class	2	4	+2	100%
2	Extend class with attributes	5	4	-1	-20%
3	Extend class with one operation	5	4	-1	-20%
4	Modify an attribute	5	10	+5	100%
5	Duplicate a class	1	11	+10	1000%
6	Add an aggregation	4	2	-2	-50%
7	Modify an aggregation	1	5	+4	400%
8	Change a generalization	1	5	+4	400%
9	Delete a class	2	3	+1	50%
10	Undo class delete	2	2	+/-0	0%
11	Simple class diagram	39	48	+9	23%
12	Simple sequence diagram	4	29	+25	625%
13	Change message direction	3	11	+8	266%
14	Change message type	3	5	+2	66%
15	Add a message	3	4	+1	33%
16	Create a sequence diagram	12	49	+37	308%
			Median Relative Difference		83%

# UI = number of user interactions

Table 7.1: Results

and specialized applications. The emerging graphical user interfaces tried to approach all processes using graphical metaphors using mouse and icons, etc. But there are tasks that can be done more efficiently using text-based input. Like searching for files or applications—the graphical metaphor provides large amounts of icons that have to be deciphered by the user.

OS X and Windows Vista introduced a text-based desktop search. The icons are filtered by the user's text input and delivers a significantly faster search process. The process happens in realtime—by typing the search string, every keystroke refines the result, allowing the user to react instantly by stopping the search or correcting the search string if necessary.

Lately Google introduced its *Instant*[2] search which features the same behavior in it's online web search interface. Google states that users save about 2-5 seconds by getting faster results and reduced numbers of required search string characters. See figure7.1.

Mobile devices like PMPs (personal media player) and mobile phones feature text-based instant search applications as well. Android and iOS devices can search their contacts lists by typing names or numbers and instantly reduce the result list iteratively by every key stroke—thus reducing the necessary user input significantly.

LaTeX is another good example of a broadly accepted text-based interface. Although there is a number of GUI front ends, its power and simplicity of use is based on the simple markup language.

[2]http://www.google.com/instant/

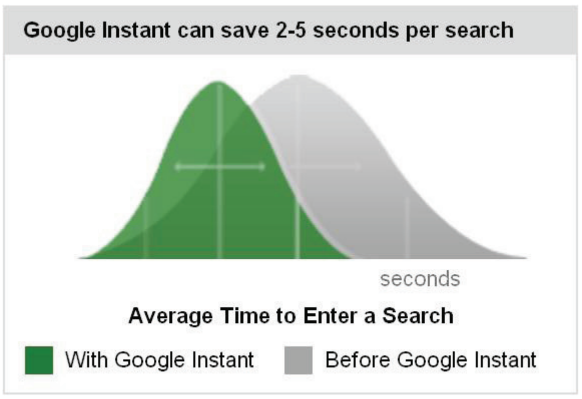


Figure 7.1: Google Instant

Tools have to be able to both tackle complex requirements, and provide intuitive base functionality.

7.3 Development progress of UML tools

After the acquirement of Rational by IBM, Rational Rose has been integrated into IBM's software development suites like Visual Age (which was renamed to Rational Software Architect (RSA) and later on to the Rational Software Delivery Platform (Rational SDP)). These tools are heavy weight software development suites which bind the user to processes and standards, like the unified rational process. Features like round-trip engineering demand strict standards conformity, and thus make the modeling process slow and cumbersome. The past years development show that round-trip engineering was not as widely accepted as predicted. Several small tools have emerged that are specialized to well-defined domains. DSLs (domain specific languages) and MDSD (model driven software development) tools, some using UML notation, compete successful against Rational Rose.

The field of computer science education requires lightweight tools like UMLet, Violet or UML Pad. These small tools are simple to deploy, simple to learn, and simple to use as these tools do not enforce strict standards but leave space for the process of creative and relaxed modeling.

7.4 End user experience

According to the mails incoming from all over the world UMLet is being used by software developers and systems analysts in diverse companies and organizations like IEEE (Institute of Electrical and Electronics Engineers), Northrop Grumman Corporation, Debian Linux and NASA (National Aeronautics and

Space Administration).

In the following there are some user opinions on UMLet:

> "I'm battle weary from UML 2.0 and Poseidon. Its great to see an open source system that is so easy to use – without all the fluff that is unnecessary."
>
> "I really have started enjoying doing UML since using UMLet. I won't argue it's full-featured like the $1000 tools out there, but it gives me 80% of the functionality for 20% of the work."
>
> "I'm currently teaching a design patterns class to a group of 16 developers in Denver, CO in the U.S. I'm having them use your UMLet tool to do all of their diagrams and it is working wonderfully. They appreciate the simplicity of the tool."
>
> "In one of the first lectures I took at university, we were taught UML. For the practical part of the lecture we had to draw several diagrams. They recommended to use Rational Rose for this purpose. Since I thought they would recommend the best software tool for us, I started to use it. But soon I was very frustrated because of the time I took to learn it. Sure such a tool is very powerful, but the purpose of learning UML should not be to learn to command the software application. In my 5th term I discovered UMLet which, I think is the very tool to for learning UML, since one is not distracted by all the option of a big grown application. It does what you expect: drawing UML diagrams. Someone can start to use it immediately. Even on Linux, which I happen to find very useful, since not all of us use the Microsoft operating systems."
>
> "This is a great tool and I find it easier to teach my students UML diagrams."
>
> "I am currently in a course that requires me to draw UML class diagram. I came across your website and and find umlet very suitable for the use for my course."
>
> "Of all the tools that I reviewed, yours stands for its simplicity and ease of use. The concept of an user interface without any popups is simply genius."
>
> "Your tool is how they all should work , doing work for me instead of making me totally frustrated and sucking all my energy out of my body and brain. I' have been looking for something that works for me for several days now and it is all crap out there. All that marketing talk makes me sick. Thanks for the enlightment."

7.5 A critical look at UMLet's present problems

Taking a closer look at UMLet naturally uncovers some problems that have been carried through some versions by the fact of the limited man power at the development of the tool.

There are some inconsistencies at the syntax in different elements. To simplify the usage there should be a consolidation of the element's syntaxes and grammars.

Another issue might be UMLet's graphical appearance. Since it is intended to be used for education, and students like software that's looking cool more than old style user interfaces, UMLet might need a brush up on its appearance. But this must be done carefully so as not to displease users that are used to UMLet.

7.6 UMLet's future

UMLet's further development will deal with the following issues:

- Consolidate the syntax of all elements to fix some inconsistencies between the element's syntax.
- Automatic resize feature of class elements.
- Movement of entire arrows with one single drag and drop action.
- Further improvements of UMLet's user interface.

There are plans to make UMLet web based using a SVG (Scalable Vector Graphics) interface. Users could discuss and share diagrams online. There could be a UML community with users helping each other.

Chapter 8

Summary and Future Work

8.1 Summary

For the quick reader: The proof of the hypothesis is discussed in section *8.1.6 Quantitative Evaluation* on page 131.

8.1.1 UML

The UML is a standardized modeling language controlled by the OMG that has developed from different methods of describing systems and requirements. After the software industry recognized UML's potential the UML Partners consortium was joined by a number of well known companies representing broad acceptance. The UML provides a set of tools to describe a system and it's details. The UML 2.0 standard features 13 different diagram types specialized to cover the respective structural and behavioral aspects of the system. UML is not like strictly constrained law, but it is an agreed standard that provides notations allowing the users to adapt in a way that may ease the understanding of what is meant.

8.1.2 Design Patterns

Design patterns help to understand recurring patterns of software design. The idea behind design patterns is to present solutions for common design problems helping the developers not to reinvent the wheel again and again, to assist discussing on a higher level and to help understanding existing software systems. Design patterns are identified design solutions to common recurring problems. Design patterns are not finished and ready to implement solutions but are a description how to solve a class of problems.
UMLet is designed to allow users to extend its functionality for their needs not only by being written using Java as the programming language. The emphasized use of design patterns helps users that want to extend UMLet understand how

it is structured and how the components work together. Design patterns help one to understand the existing design as fast as possible.

8.1.3 Requirements for lightweight UML tools

Simple and understandable lightweight UML tools reduce complexity and functionality to the basic needs. The UML provides a graphical notation that invites to sketch and play with the model in early design stages. Thus lightweight UML tools must cover the requirements *easy to use*, *easy to learn*, *easy to deploy*, and provide *relaxed standards restrictions*.

While *easy to use* and *easy to learn* aim at simple user interfaces and simple workflows, *easy to deploy* concerns the fact that the process of distribution and deployment may be critical—especially if the tool is used for educational purposes, and students are required to install the software on their home computers. The requirement for *relaxed standards restrictions* applys to explorative sketching. This is the process of creating not necessarily exact and complete diagrams. Tools that enforce strict standards conformity slow down the design process and draw too much attention to the software tool itself, than to the creation and development progress.

8.1.4 UMLet's agile approach

UMLet is such a lightweight UML sketching tool. In order to provide a lightweight tool UMlet's design focuses on concepts like *unobtrusiveness* by avoiding unnecessary distractions like pop-ups, *fast diagram creation* providing a text-based modeling approach, *easy expandability* featuring dynamic custom-elements, and *independence from operating systems* by using a 100% Java implementation.

8.1.5 Competitors and Features

Chapter 4 presents UMLet's direct competitors like ArgoUML, Violet and Dia. But since Rational Rose is the de-facto industry standard of UML tools, all other serious competitors have to match with it. One main feature of Rational Rose is its round-trip engineering functionality which allows to generate source code from UML models and vice versa. Thus requiring the models to be strictly UML standards conform, reducing the comfort of the model creation process. UML tools usually treat UML elements as visual objects, whose appearance can be edited by changing their attributes. This is mostly done through pop-up dialog boxes. Rational Rose's dialog to edit a UML class element and its attributes contains 8 tabs, and approximately 40 user interface elements.

UMLet's approach to modeling is different. It provides a simple user interface without distracting dialogs and pop-up windows, or hardly decipherable icons. UMLet offers simple element palettes presenting the real, full-sized elements in context of usage. Fast diagram creation is achieved by text-based modeling featuring powerful grammar to create and modify elements and diagrams.

8.1.6 Quantitative Evaluation

The research hypothesis presented in chapter 6 was that *text-based user input can reduce the number of user interactions with a graphical user interface, increase the speed of user input and lead to a more efficient user interface.*

To prove the hypothesis and show the potential of text-based modeling the assessment features a quantitative comparison method. Users with different experience levels judge differently on user interfaces. Raskin [36] provides quantifiable metrics to assess the quality of user interfaces. The GOMS methods measure user interfaces relying on 6 primitive operations (pressing a key, moving the mouse, dragging the mouse, mental preparation, moving hands, and waiting for command execution). The applied simplified approach concentrates on mouse clicks and combined keyboard inputs, while disregarding mouse movements or individual key presses.

The evaluation features 16 common use cases that are essential to UML modeling. The result of the evaluation shows that users of Rational Rose are required to perform about 80% more interactions. This is an achievement of UMLet's design goals of *unobstrusivness* by avoiding unnecessary pop-ups and providing a simple and monotonous user interface, and *fast diagram creation* featuring text-based element creation. The result proves the hypothesis that text-based user input can increase the efficiency of user interfaces. Use case 16 demonstrates UMLet's power of grammar-based diagram creation. The creation of sequence diagrams by textual descriptions is extremly efficient in comparison to conventional user interface approaches.

8.2 Future Work

As the evaluation proved, UMLet's text-based approach is very efficient. Other applications of the text-based approach show its potential, like searching for files or applications, as OS X's or Windows Vista's desktop searches. The conventional graphical metaphor providing large amounts of icons is replaced by a text-field where each keystroke is instantly applied on the search result.

UMLet has a sister project named PLOTlet[1]. The text-based approach is vertically expanded to the use of plots like bars, lines, and pies. Changing the type of a plot is as simple of replacing a string in the textual description, as

[1]www.plotlet.com

demonstrated by UMLet.

Some tools offer an expandability possibility by end-user programming. They offer macros with limited language or API support, or plug-in development suffering from a media break since the implementation is done externally. UMLet demonstrates EUP [5] by integrating a run-time Java compiler within the application. A well commented template, code assistance, and a preview pane, make expanding UMLet simple.

Chapter 9

Appendix

9.1 Template file for custom elements

```
// Some import to have access to more Java features
import java.awt.*;
import java.util.*;
import com.umlet.control.*;
import com.umlet.element.base.Entity;

public class NewElement extends com.umlet.element.base.Entity {

  // Change this method if you want to edit the graphical
  // representation of your custom element.
  public void paint(Graphics g) {

    // Some unimportant initialization stuff; setting color, font
    // quality, etc. You should not have to change this.
    Graphics2D g2=(Graphics2D) g; g2.setFont(Constants.getFont());
    g2.setColor(_activeColor); Constants.getFRC(g2);


    // It’s getting interesting here:
    // First, the strings you type in the element editor are read and
    // split into lines.
    // Then, by default, they are printed out on the element, aligned
    // to the left.
    // Change this to modify this default text printing and to react
    // to special strings
    // (like the "--" string in the UML class elements which draw a line).
    Vector tmp=Constants.decomposeStrings(
                              this.getPanelAttributes(), "\n");
  int yPos=Constants.getDistLineToText();
  for (int i=0; i<tmp.size(); i++) {
    String s=(String)tmp.elementAt(i);
    yPos+=Constants.getFontsize();
    Constants.write(g2,s,Constants.getFontsize()/2, yPos, false);
    yPos+=Constants.getDistTextToText();
  }


  // Finally, change other graphical attributes using
  // drawLine, getWidth, getHeight..
  g2.drawLine(0,0,this.getWidth()-Constants.getFontsize(),0);
  g2.drawLine(this.getWidth()-Constants.getFontsize(),0,
              this.getWidth()-1, Constants.getFontsize());
  g2.drawLine(this.getWidth()-1, Constants.getFontsize(),
              this.getWidth()-1, this.getHeight()-1);
  g2.drawLine(this.getWidth()-1, this.getHeight()-1,
              0, this.getHeight()-1);
  g2.drawLine(0, this.getHeight()-1, 0, 0);
  g2.drawLine(this.getWidth()-Constants.getFontsize(),
              0,this.getWidth()-Constants.getFontsize(),
              Constants.getFontsize());
  g2.drawLine(this.getWidth()-Constants.getFontsize(),
              Constants.getFontsize(),this.getWidth()-1,
              Constants.getFontsize());
}


// Change this method if you want to set the resize-attributes of
// your custom element
public int getPossibleResizeDirections() {
  // Remove from this list the borders you don’t want to be resizeable.
  return Constants.RESIZE_TOP | Constants.RESIZE_LEFT |
         Constants.RESIZE_BOTTOM | Constants.RESIZE_RIGHT;
}


// Advanced: change this method to modify the area where relations
// stick to your custom element.
public StickingPolygon getStickingBorder() {
  // By default, the element returns its outer borders. Change it,
  // if your element needs to stick to relations differently.
  // See, for example, the source code of the UML interface element.
  StickingPolygon p = new StickingPolygon();
  p.addLine(new Point(0,0), new Point(this.getWidth()-1,0),
            Constants.RESIZE_TOP);
    p.addLine(new Point(this.getWidth()-1,0),
              new Point(this.getWidth()-1,this.getHeight()-1),
              Constants.RESIZE_RIGHT);
    p.addLine(new Point(this.getWidth()-1,this.getHeight()-1),
              new Point(0,this.getHeight()-1),Constants.RESIZE_BOTTOM);
    p.addLine(new Point(0,this.getHeight()-1), new Point(0,0),
              Constants.RESIZE_LEFT);
    return p;
  }
}
```

List of Figures

Bibliography

[1] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns, Best Practices and Design Strategies, Second Edition*. Prentice Hall, 2003.

[2] Apple Computer Inc. *Macintosh Human Interface Guidelines (Apple Technical Library)*. Addison Wesley, 2nd edition, 1992.

[3] E. Astesiano and G. Reggio. UML-spaces: a UML profile for distributed systems coordinated via tuple spaces. In *Proceedings of the 5th International Symposium on Autonomous Decentralized Systems, 2001*, pages 127–134, 26-28 March 2001.

[4] Martin Auer, Ludwig Meyer, and Stefan Biffl. Explorative uml modeling - comparing the usability of uml tools. In *Proceedings of the 9th International Conference on Enterprise Information Systems (ICEIS 2007)*, pages 466–474. ICEIS, INSTICC Press, 2007.

[5] Martin Auer, Johannes Poelz, and Stefan Biffl. End-user development in a graphical user interface setting. In Jose Cordeiro and Joaquim Filipe, editors, *Proceedings of the 11th International Conference on Enterprise Information Systems*, pages 5–14, LNBIP 24, 2009. Springer.

[6] Martin Auer, Thomas Tschurtschenthaler, and Stefan Biffl. A flyweight UML modeling tool for software development in heterogeneous environments. In *Proceedings of EUROMICRO 2003, Antalya*, 2003.

[7] Martin Auer, Thomas Tschurtschenthaler, and Stefan Biffl. A flyweight uml modelling tool for software development in heterogeneous environments. Technical report, Institute of Software Technology Vienna University of Technology, 2003.

[8] Wendy Boggs and Michael Boggs. *Mastering UML with Rational Rose 2002*. Sybex, 2002.

[9] Grady Booch, James Rumbaugh, , and Ivar Jacobson. *Unified Modeling Language User Guide, The*. Addison Wesley, 2nd edition, 2005.

[10] Jung-Wei Chen and Jiajie Zhang. Comparing text-based and graphic user interfaces for novice and expert users. *AMIA Annual Symposium Proceedings*, pages 125–9, 2007.

[11] Qi Chen, J. Grundy, and J. Hosking. An e-whiteboard application to support early design-stage sketching of UML diagrams. In *Proceedings of the IEEE Symposium on Human Centric Computing Languages and Environments, 2003*, pages 219–226, 28-31 Oct. 2003.

[12] James W. Cooper. *The Design Patterns Java Companion.* Addison Wesley, 1998.

[13] H. Dagdeviren, R. Juric, and P. Lees. Experiences of teaching UML within the information systems curriculum. In *26th International Conference on Information Technology Interfaces, 2004*, volume 1, pages 381–386, 2004.

[14] Berthold Daum. *Professional Eclipse 3 for Java Developers.* Wrox, 2004.

[15] Harvey M. Deitel, Paul J. Deitel, and Sean E. Santry. *Advanced Java 2 Platform: How to Program.* Prentice Hall, 2001.

[16] H. Eichelberger. Evaluation-report on the layout facilities of UML tools. Technical report, Department of Computer Science, University of Wuerzburg, 2002.

[17] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado. *UML2 Toolkit.* Wiley Publishing, 2004.

[18] A. S. Evans and A. J. 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 (Ref. No. 1999/006)*, pages 2/1–2/4, Jan. 1999.

[19] Martin Fowler. Patterns. *IEEE Software*, March/April, 2003.

[20] Martin Fowler. *UML Distilled: a brief guide to the standard object modeling language.* Addison Wesley, 3rd edition, 2003.

[21] Kurt A. Gabrick and David B. Weiss. *J2EE and XML Development.* Manning, 2002.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[23] Martin Glinz. The teacher: 'concepts!' the student: 'tools!' — on the number and importance of concepts, methods, and tools to be taught in software engineering education. In *Proceedings of the Third International Workshop on Software Engineering Education*, pages 32–34 and 55. Softwaretechnik-Trends, 1996.

[24] Timothy J. Grose, Gary C. Doney, and Stephen A. Brodsky PhD. *Mastering XMI: Java Programming with XMI, XML, and UML.* Wiley, 2002.

[25] Klaus Marius Hansen and Anne Vinter Ratzer. Tool support for collaborative teaching and learning of object-oriented modeling. In *Proceedings of the 7th annual conference on Innovation and technology in computer science education*, ITiCSE '02, pages 146–150, New York, NY, USA, 2002. ACM.

[26] Elliotte Rusty Harold. *XML 1.1 Bible, 3rd Edition.* Wiley, 2004.

[27] IEEE. IEEE standard computer dictionary: A compilation of ieee standard computer glossaries. In *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. IEEE, 1990.

[28] H.J. Kohler, U. Nickel, J. Niere, and A.Zundorf. Integrating UML diagrams for production control systems. In *Proceedings of the 2000 International Conference on Software Engineering, 2000*, pages 241–251, 4-11 June 2000.

[29] 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, 2003*, pages 227–235, Oct. 2003.

[30] Dean Leffingwell and Don Widrig. *Managing Software Requirements: A Use Case Approach.* Addison Wesley, 2nd edition, 2003.

[31] Floyd Marinescu. *EJB Design Patterns, Advanced Patterns, Processes, and Idioms.* Wiley Publishing, 2002.

[32] N. Medvidovic, A. Egyed, and D. S. Rosenblum. Round-trip software engineering using UML: From architecture to design and back. In *Proceedings of the 2nd Workshop on Object-Oriented Reengineering (WOOR), 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Toulouse, France*, Sept. 1999.

[33] Thomas P. Moran, Patrick Chiu, and William van Melle. Pen-based interaction techniques for organizing material on an electronic whiteboard. In *Proceedings of UIST 1997, Banff, Alberta*, pages 105–114, 14–17 October 1997.

[34] Anthony Potts and David H. Friedl Jr. *Java Programming Language Handbook.* Coriolis Group, 1996.

[35] D. Quan, D. Huynh, Dr. Karger, and R Miller. User interface continuations. In *Proceedings of the 16th annual ACM Symposium on User interface software and technology. Vancouver, Canada*, pages 145–148, 2003. ISBN:1-58113-636-6.

[36] Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems.* Addison Wesley, 2000.

[37] Erik T. Ray. *Learning XML.* O'Reilly & Associates, 2001.

[38] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 2 edition, 2004.

[39] Douglas C. Schmidt and Paul Stephenson. Experience using design patterns to evolve communication software across diverse os platforms. Technical report, Department of Computer Science Washington University St. Louis, MO 63130 and Ericsson Inc. Cypress, CA 90630, 1990.

[40] Bruce Tate. *Bitter Java*. Manning, 2002.

[41] J. Tenzer. Improving UML design tools by formal games. In *Proceedings of the 26th International Conference on Software Engineering, 2004. ICSE 2004*, pages 75–77, May 2004.

[42] The Object Man Group. www.omg.org, 2006.

[43] Jenifer Tidwell. *Designing Interfaces*. O'Reilly Media, 2005.

[44] Scott A. Turner, Manuel A. Pérez-Quiñones, and Stephen H. Edwards. minimuml: A minimalist approach to uml diagramming for early computer science education. *J. Educ. Resour. Comput.*, 5, December 2005.

[45] Bin Zhang and Ye sho Chen. Enhancing UML conceptual modeling through the use of virtual reality. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences, 2005. HICSS '05*, page 11b, Jan. 2005.