FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

A Legacy System Migration and Software Evolution

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Andreas Fürnweger
Matrikelnummer 0626472

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. DI Mag. Dr. Stefan Biffl
Mitwirkung: Dr. DI Mag. MA Martin Auer

Wien, 20.04.2017

(Unterschrift Verfasser) (Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Andreas Fürnweger
Industriestraße 135/1/29, 1220 Wien

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

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Software ages. Maintenance costs tend to increase, and modifications to an application make future adaptions more difficult. As the surrounding software components are updated and modernized, static software becomes even more outdated relative to them.

As soon as a system notably resists modification and evolution, it becomes legacy software and the stakeholders, such as architects and managers, have to decide whether to preserve or redesign the system.

The main research questions are:

- What software evolution approaches are feasible and how to evaluate their cost and risk criteria?
- How do these criteria affect an actual migration based on a large, real-world software package?

The chosen evaluation methods are:

- Research software evolution and related topics to identify different evolution approaches, and create a list of software evolution criteria for them.
- Apply those criteria to a real-world application to find an appropriate evolution approach; break it down to milestones; implement and evaluate the success of the implementation.

The results are:

- The evolution can be preservation or migration driven; many offsetting costs/benefits and risk/reward profiles must be considered.
- As for real-world instances of migrations, there exist several tools to ease migrations and enable cross-platform application development. A code analysis is a useful way to quantify the success of the implementation.

This work is the extended version of the paper *Software Evolution of Legacy Systems: A Case Study of Soft-Migration* [38].

Kurzfassung

Software altert. Die Wartungskosten steigen tendenziell an und Änderungen an einer Anwendung machen zukünftige Anpassungen schwieriger. Wenn Softwarekomponenten im Umfeld aktualisiert und modernisiert werden, wirkt statische Software relativ dazu noch stärker veraltet.

Sobald sich die Änderung und Weiterentwicklung eines Systems deutlich erschwert, bezeichnet man es als Legacy System (Altsystem). Interessengruppen wie Architekten und Manager müssen dann entscheiden, ob das System präserviert oder redesigned werden soll.

Die wichtigsten Forschungsfragen sind:

- Welche Ansätze zur Software Evolution sind machbar und wie kann man die Kosten- und Risikofaktoren beurteilen?
- Wie beeinflussen diese Faktoren eine tatsächliche Migration basierend auf einem umfangreichen, realen Softwarepaket?

Die gewählten Bewertungsverfahren sind:

- Software Evolution und verwandte Themengebiete recherchieren, um verschiedene Evolutionsansätze herauszuarbeiten; daraus eine Liste an möglichen Software Evolutions Kriterien erstellen.
- Diese Kriterien auf eine reale Softwareanwendung umlegen, um einen passenden Evolutionsansatz zu finden; auf Meilensteine herunterbrechen; implementieren und den Erfolg der Implementierung beurteilen.

Die Ergebnisse sind:

- Die Evolution kann in Richtung Präservierung oder Migration gehen; das Kosten/Nutzen und das Ertrags/Risiko Verhältnis diverser Ansätze muss beachtet werden.
- Es gibt viele Werkzeuge, um die Migration von realen Softwareprojekten zu erleichtern und um plattformübergreifende Anwendungen zu entwickeln. Eine Codeanalyse ist ein praktischer Weg, um den Erfolg der Umsetzung zu messen.

Diese Diplomarbeit ist die erweiterte Fassung des Papers *Software Evolution of Legacy Systems: A Case Study of Soft-Migration* [38].

Contents

CHAPTER 1

Introduction

1.1 Motivation

Software development is still a fast-changing environment, driven by new and evolving hardware, operating systems, frameworks, programming languages, and user interfaces. While this seemingly constant drive for modernization offers many benefits, it also requires dealing with legacy software that—while working—slowly falls out of step with the surrounding components that are being updated—for example, if a certain version of an operating system is no longer supported by its vendor.

There are various ways to handle such “aging” software: one can try to keep it up and running; to carefully refactor it to various degrees to make it blend in better; to port its code; to rewrite it from scratch; etc. The main stakeholders in deciding on a course of action are managers, which must allocate resources to and consider the risks and maintenance cost profiles of the various options (e.g., will affordable developers with specific skills still be available?), as well as software developers, which should be aware of the long-term implications of their choices (e.g., will a certain programming language be around in five years’ time?).

To provide some software evolution guidelines, this thesis gives an overview of the state of the art of software evolution, as well as maintenance, reengineering, and whether to preserve or redesign legacy systems.

It looks into different aspects of software maintenance and shows that the classic meaning of maintenance as some final development phase after software delivery is outdated. Instead, it is best seen as an ongoing effort.

Then costs and benefits of various evolution approaches are outlined. These approaches are either legacy-based, essentially trying to preserve as much as possible of the existing system, or migration-based, where the software is transferred, to various degrees, into a new setup.

After that, so-called “soft” migration approaches are discussed. Those approaches aim to facilitate traditional migration methods like porting or rewriting code via support tools and frameworks.

Special focus is given to the Java programming language and a specific variant of a soft-migration approach is presented, which is using a Java-based program core with several platform-specific branches. The approach is explained in detail, starting with its scope, the basic idea, a step by step execution, and a description of tools that can greatly facilitate the migration effort.

Finally, a case study of an actual soft migration is described and the benefits and drawbacks of the suggested approach are discussed.

The results are also published in the paper **Software Evolution of Legacy Systems: A Case Study of Soft-Migration** [38].

1.2 Research Questions

The two main research questions of this thesis are:

- (R1) What software evolution approaches are feasible and how to evaluate their cost and risk criteria?

R1 is addressed by a comprehensive literature study in the scientific fields of software evolution and maintenance. The goal is to extrapolate different evolution approaches and create a list of software evolution criteria for them, to assist the decision-making process of the stakeholders, such as managers and software engineers, who have to decide whether to preserve or migrate a specific piece of software.

- (R2) How do these criteria affect an actual migration based on a large, real-world software package?

R2 is addressed by applying the results of the previous research to an actual software package. Based on those software evolution criteria, an appropriate way to evolve the program should be found, and the results should be quantified, evaluated and discussed.

1.3 Actual Migration

To provide an exemplary soft migration of a large, real-world software project, the UML sketching tool UMLet[1] has been chosen.

The first version of UMLet has been released on 2002-06-12. Since then many students have worked on the codebase and a large portion of the diagram elements are submissions from different volunteers which helped to create the wide area of elements which UMLet has to offer.

Over the last years, many new features have been introduced to UMLet, like all-in-one diagrams, custom elements, class diagram generation from Java source code, plot generation using the spin-off project Plotlet, . . .

With all these new features, the program evolved from a simple UML sketching tool to a generic tool which interprets properties of a collection of diagram elements and afterwards draws a graphical representation of them.

[1]`http://umlet.com/`

The problem is, 10 years ago nobody could foresee that UMLet will get that popular and will still be extended with new features. Therefore most of the features are based on a code base which has not been designed with such flexible usage in mind.

The first steps to accommodate UMLet's evolution to a more flexible tool have been evaluated and implemented accompanying a bachelor's thesis [37]. These changes extended the elements from mostly UML based diagrams to various kinds of plots.

This thesis focuses on the platform portability of UMLet. The program is currently available for all Java platforms as a Swing-based standalone application, and as SWT-based Eclipse plugin. The soft migration described in this thesis migrates the program to a native web application, which extends the availability to any platform with a web browser.

Milestones

The main goal of the reengineering process is to make UMLet's codebase more flexible and modular than it is today, and to migrate the application to the web. Therefore the following milestones should be achieved:

- (M1) Project Modularization: Modularize the application by separating platform specific code from the shared code.

- (M2) Web Version Implementation: Design and implement a new web version of UMLet, while still maintaining the standalone and eclipse plugin versions.

1.4 Stakeholders

In large software projects, there are many different stakeholders, who have an interest in the way the system evolves.

- **Managers** have to allocate resources and manage the risks of legacy code or migrations. They also have to hire people, which might be difficult or impossible in case of now rarely used programming languages like COBOL.

- **Software architects** have to design systems that communicate with or incorporate legacy systems or the have to design a new architecture if the software should be migrated.

- **Programmers** have to learn new programming languages and tools to either maintain the legacy software or migrate it to new platforms.

- **Customers** express preferences and request features which influence the direction a software evolves. For example they probably also want to use the software with their smartphones or with a web browser.

- **Market analysts** have to analyze the market and potential customer wishes, to evaluate which new platforms are future proof or which new features are required.

1.5 Related Work

Mens and Demeyer [61] give an overview of trends in software evolution research and address the evolution of software artifacts like databases, software design, and architectures.

A general overview of the related topics of maintenance and legacy software is given by Bennett and Rajlich [11], who also identify key problems and potential solution strategies. According to them, software evolution is an interdisciplinary problem to be tackled from a business and a technological angle.

Lehman classifies programs in terms of software evolution and also formulates eight laws of software evolution [56, 57], which are, however, not considered universally valid according to Herraiz et al. [43].

There are also many exploratory studies which try to analyze and understand software evolution based on specific software projects [18, 50, 52, 77, 95].

Chaikalis and Chatzigeorgiou develop a prediction model for software evolution and evaluate it against several open-source projects [19]. Benomar et al. present a technology to identify software evolution phases based on commits and releases [12].

The related topic of legacy systems is a bit ambiguous, due to several existing definitions of the term. It can describe a system which resists modification [14], a system without tests [32], or even all software as soon as it has been written [45].

A natural question regarding legacy systems is whether to preserve or redesign them. As this question is not easy to answer [85], the pros and cons of reengineering or preserving a system are to be compared thoroughly before making a decision [89]. In addition, it is possible to replace a system in stages to minimize the operational disruption of the system [85].

Due to the inevitable process of software aging [72], the maintenance costs of an aged application increase every year and the program gets harder to modify.

Even though the classic view of maintenance as the final life-cycle phase of software after delivery [82] is still prevalent, it's a much broader topic, especially for programs which must constantly adapt to a changing environment.

There are reports that total maintenance costs are at least 40% of the initial development costs [15], 70% of the software budget [42], and up to 90% of the total costs of the system [76]. As these numbers show, the topic of maintenance is crucial. Lientz and Swanson [59] categorize maintenance activities into classes. Several authors [79, 88] propose maintainability metrics.

Finally, when migrating a system, a reengineering phase is almost always necessary. According to Feathers [32], this phase should be elaborate and accompanied by extensive testing to make sure the application behavior stays the same. Gottschalk et al. [40] show that besides migration, there are also other reasons to reengineer software (e.g., to reduce the energy consumption on mobile devices).

Fowler and Beck [36] list useful refactoring patterns, while Feathers [32] stresses how legacy code can be made testable.

Cross-Platform application development is a much-discussed topic in the scientific community [28, 44, 80, 81]

1.6 Structure of the Work

The structure of the work can be separated into several chapters.

Chapter 2 starts with an overview of software evolution and related topics, such as software aging and maintenance. It continues with a listing of several software evolution approaches, and defines a list of software evolution criteria to support finding the correct decision for a specific application. It concludes with a discussion of soft migration approaches, with a special focus on the Java programming language.

After elaborating the theoretical background, the state of the art of UMLet's environment is discussed in chapter 3. It describes the increase in portability of software, especially Java based software, and focuses on the web as a portable modern platform. GWT is described as a way to bring Java code to the web, and other web based UML tools are analyzed, which may inspire the design of UMLet's web version.

Chapter 4 applies the software evolution criteria to UMLet, to find an appropriate evolution approach. It also describes the high-level architectural model of UMLet, which then gets improved, in order to make the core logic of UMLet portable. Finally a web platform specific architectural model is created.

The implementation of the first milestone (M1) is presented in chapter 5, which describes the necessary steps to modularize UMLet in preparation to migrate to new platforms. It discusses reusable functions, a generic properties parser and a generic drawing API, which can be implemented by different platforms. Finally, the separation of the Eclipse projects and the results of the migration are presented.

Chapter 6 describes the creation of the web version UMLetino, which is the second milestone (M2). It introduces the general idea of reusing most Java code using a transpiler (GWT), discusses the UI design goals, and describes several important steps such as the first prototype and iterative improvements of the UI, concluding with the finished UMLetino version.

The success of the chosen reengineering approach is evaluated in chapter 7, by comparing statistics of the codebase before and after the migration, such as findbugs warnings, lines of code and the general project and package dependency structure.

Chapter 8 describes the adaptation of the build tool Apache Maven, which improves the build process and maintainability of UMLet's modular code base.

Chapter 9 discusses the results, by answering the research questions, and by describing the implementation of the milestones.

Finally, chapter 10 presents potential future improvements to the general codebase, as well as standalone and web version specific enhancements.

CHAPTER 2

Software Evolution

Software evolution and maintenance are integral parts of the modern software life cycle. Some programs only get minor bug-fixes from time to time (e.g., for security reasons), but many tools constantly evolve by introducing new features, reacting to user feedback and adapting to changes in the environment.

The following sections give an overview of software evolution and discusses related topics, such as the effects of software aging, legacy systems and what to do with them, and the importance of maintenance, which should not be treated as just the final phase of the software life-cycle.

Afterwards, different software evolution approaches are shown, which may lean more towards preservation or migration. Finally, a criteria list is presented to help stakeholders, who have to weigh cost/risk profiles of different evolution strategies, to find the best strategy for a specific piece of software.

2.1 Software Evolution as Scientific and Engineering Discipline

As described by Lehman et al. [58], Software Evolution can be distinguished into two types:

1. Software Evolution as a **scientific discipline** with the focus on the **what**: investigate its nature and impact.

2. Software Evolution as an **engineering discipline** with the focus on the **how**: investigate its pragmatic aspects (technology, methods and tools to effectively and reliably evolve a software system).

Software Evolution as a Scientific Discipline

There are several ways to categorize software in terms of Software Evolution.

Lehman's categories of software and laws

One popular classification of programs regarding Software Evolution has been done by Meir M. Lehman in September 1980 [56]. In this article he classifies programs according to their relationship to the environment in which they are executed:

- **S-Program:** It's function is formally defined by a specification. This specification can also relate to the external world, e.g., the traveling salesman problem.
- **P-Program:** They should solve real world problems, where precise problem statements are not possible and approximations and feedback loops are part of the program, e.g., a chess program.
- **E-Program:** In addition to the feedback loop, it must constantly adapt to changing requirements and circumstances in their environment, e.g., an operating system or air-traffic control.

Lehman is also known for his **laws of software evolution**. All eight laws are listed by Lehman et al. [57] and they describe typical behaviors of apply to E-type systems. The first two laws are:

- **Continuing Change:** An E-type system must be continually adapted or it becomes progressively less satisfactory [57].
- **Increasing Complexity:** As an E-type system evolves, its complexity increases unless work is done to maintain or reduce it [57].

ISO/IEC Standard 14764:2006

An alternative approach is the categorization by types of maintenance which is done in the ISO/IEC Standard 14764:2006 [47]:

- **Adaptive maintenance:** modifications performed after delivery to keep the product usable in changing environment.
- **Corrective maintenance:** modifications performed after delivery to correct discovered problems.
- **Perfective maintenance:** modifications performed after delivery to detect and correct faults before they manifest as failures.
- **Preventive maintenance:** modifications performed after delivery to detect and correct latent faults before they become operational faults.

Software Evolution as an Engineering Discipline

There are many exploratory studies about software evolution in different areas, like open-source software [50], eclipse plugins [18] and mobile apps [95].

Ratzinger et al. [77] try to analyze the development history and predict locations of future refactoring, and Kin et al. [52] relate the amount of and time needed for bug fixes to refactoring efforts during software evolution.

This thesis also focuses mostly on the practical aspects of software evolution and presents a possible way to evolve a real-world non-trivial software package.

2.2 Software Aging

"Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limits its effects ... and prepare for the day when the software is no longer viable." [72]

Parnas mentions two types of software aging:

1. **Failure to modify the program to meet changing needs:** programming languages, systems which run the program and user expectations change. As a result, if those changes are not considered during the software lifecycle, users will switch to better alternatives.

2. **The result of the changes that are made:** Aging can be the result of changes, if the programmers are not familiar with the initial software architecture or concept. They can introduce changes which are inconsistent with the original design, which makes the program harder and more expensive to maintain and understand.

The maintenance costs of an aged application tend to increase, because modifications to a software generally make future adaptions more difficult. Therefore it is important to invest time to keep software modules simple, to clean up convoluted code, and to redesign program logic if necessary [64].

2.3 Legacy Systems

There are several different definitions of what a legacy system or legacy code is.

According to Merriam-Webster [62], a legacy system is a previous or outdated computer system.

Brodie and Stonebraker [14] describe it as *"a system which significantly resists modification and evolution"*.

Even though these definitions are probably more in line with the common understanding of what a legacy system is, there are also some alternative ones like the one.

Feathers [32] defines it as code without tests, while Hunt and Thomas [45] say that *"All software becomes legacy as soon as it's written"*.

The topic of legacy code and what should be done with it is both a practical issue with many real world examples, and the topic of scientific research about how to solve the problems that come with such code.

Preserve or Redesign Legacy Systems

According to Schneidewind and Ebert [85], the question whether to preserve and maintain or redesign a legacy system is not easy to answer.

Both approaches have different advantages and problems and the decision is highly dependent on the specifics of the system, therefore there is no universal answer to this question.

Typical reasons to preserve a system are:

- The risk or cost of a redesign appear to be too high.
- The legacy system is flexible enough to handle future requirement changes.

Typical reasons to redesign it are:

- The stability of aging systems decrease over time and maintenance becomes more difficult and costly.
- New functionality gets more difficult to add and requires extensive regression testing.
- Developers don't understand the initial software architecture or the initial architecture was diluted over the years, therefore it's hard to evaluate the impact of small changes on the system.

In general, most organizations do not rush to replace legacy systems, because the successful operation of these systems is vital to the companies survival, but they must eventually take some action to update or replace their systems, otherwise they will not be able to take advantage of new hardware, operating systems and application programs.

An important aspect of this decision is that one must not choose an extreme solution like preserving a system unaltered or redesigning it from scratch. Instead the existing system can be maintained, while the replacement system is developed, which makes a fluid transition from the old to the new system possible.

This also has the advantage that such a replacement system can be installed in stages to minimize the disruption to the existing system and to avoid replacing the existing system as a whole while it's operational [85].

Sneed [89] remarks, that reengineering is only one of many solutions to the typical maintenance problems with legacy systems. One should compare the pros and cons of reengineering, redeveloping and doing nothing before making a decision.

He also mentions that there must be a significant benefit like cost reduction or added value to the program to justify the reengineering, and that it is important to compare the maintenance costs of the existing solution to the expected improvements introduced by the reengineering.

2.4 Maintenance

Software maintenance is sometimes considered to be the final phase of the delivery life-cycle. Unfortunately, this definition is outdated for most modern software, which must constantly adapt to changing requirements and circumstances in their environment.

Maintenance Effort and Costs

In large software code-bases, the required maintenance effort is high. Basili et al. [10] show how to build a predictive effort model for software maintenance releases, with the goal of getting a better understanding of maintenance effort and costs. Brooks [15] claims that the total maintenance costs of a widely used program are typically at least 40% of the initial development costs. Rashid et al. [76] show that over the last few decades the costs of software maintenance have increased from 35-40% to over 90% of the total costs of the system. According to Harrison and Cook [42], more than 70% of the software budget is spent on maintenance; 75% of software professionals are involved with maintenance. According to Coleman et al. [23], HP has between 40 and 50 million lines of code under maintenance, and 60% to 80% of research and development personnel are involved in maintenance activities.

These numbers illustrate that maintenance can take up large portions of the information systems budget, therefore it's important to keep aging software maintainable and to understand the different classes of maintenance to make the right decision about what to do with a legacy system.

Maintenance Classes

Lientz and Swanson [59] categorize maintenance activities into four classes: **adaptive** (keeping up with changes in the software environment); **perfective** (new functional or nonfunctional user requirements); **corrective** (fixing errors); and **preventive** (prevent future problems). The most maintenance effort (around 51%) falls into the second category, while the first category (around 23%), and the third one (around 21%), make up most of the remaining effort.

There are several metrics to evaluate how maintainable a system is. Unfortunately, these methods don't always produce consistent results [79,88]. Sjøberg et al. [88] consider the overall system size to be the best predictor of maintainability.

Reengineering

"Reengineering (...) is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring." [22]

Many times the existing software is a legacy system, although *"it is not age that turns a piece of software into a legacy system, but the rate at which it has been developed and adapted without having been re-engineered."* [25]

Depending on the architecture and documentation of the existing codebase, the reverse-engineering part will be more or less elaborate. After the program is understood, the restructuring can begin to prepare the application for the migration to a new platform.

Feathers [32] mentions that in the case of legacy systems the necessary reengineering phase has to be more elaborate and should be accompanied by the introduction of automated tests, to make sure the current application behaves the same before and after the reengineering. Gottschalk et al. [40] describe reengineering efforts to reduce the energy consumption of mobile devices.

Refactoring

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs." [36]

Typical improvements of the internal structure are: readability, reduced complexity and therefore improved maintainability.

Although some refactorings must be done manually, there are several tools and programs, often integrated into IDEs, which help developers to automate refactoring tasks. While most of those tools are language specific, the creation of language independent refactoring tools is a hot topic in science [54, 92].

Patterns

In Software Engineering, the term **Pattern** is often used to describe a solution to a certain recurring problem in the engineering process. Well known areas of patterns are Design Patterns [39, 87], Architecture Patterns [17], Analysis Patterns [35], Reengineering Patterns [25], and Testing Patterns [13]. Patterns also serve as common terminology for field experts to discuss possible solutions to certain problems.

There is also the term **Anti-Pattern** which is described by Budgen [16] as *"design anti-patterns are 'obvious, but wrong, solutions to recurring problems'."*

Patterns as well as Anti-Patterns can be very helpful when analyzing software to detect problems and possible solutions which should be applied during the refactoring phase.

2.5 Software Evolution Approaches

Simplified, software evolution comes in various flavors (in increasing order of perceived costs), and is characterized by the following activities:

Legacy-Based Evolution

1. Simple maintenance
    - Keep the system running.
    - Only apply bug-fixes and required changes.

2. Maintenance with some reengineering
    - Carefully adapt and overhaul program logic.
    - Document application logic.
    - Create automated tests if missing.

Migration-Based Evolution

3. Soft migration
    - Use tools to ease migration (e.g., virtual machines, transpilers, etc.).
    - Reuse as much as possible the core parts of the legacy source.
    - Only add minimal code in new languages (e.g., Java wrapper around existing COBOL application; HTML pages for GWT transpiled code).
4. Hard migration or porting
    - Re-program the application from scratch.
    - Re-compile existing code on new target platform.

At first glance, the costs seem to increase in this list of evolutionary steps. However, this need not be the case:

- As for (1), legacy systems set up with old programming languages (ADA, COBOL) might incur increasing maintenance costs due to a lack of available expertise.
- With respect to (4), well-programmed C-code, on the other hand, can theoretically be ported to, i.e., re-compiled on, a new operating system at almost zero cost. In practice, this very rarely happens; even supposedly platform-independent languages like Java often cause portability problems.

2.6 Software Evolution Criteria

This section summarizes benefits and risks involved in preserving or migrating software to help determine the appropriate solution for a specific application.

Preservation Benefits

- Stability (training, operations, etc.) is preserved.
- Better predictability of overall system costs (if no major changes are required).
- Saved resources can be applied to keep the software alive with minor, and less dangerous, software evolution steps than outright migration, like partial re-engineering, documentation via reverse-engineering, or virtualization.

Preservation Risks

- Legacy systems are hard to maintain and change.
- Underlying, external dependencies (e.g., hardware, operating systems, virtual machines, software frameworks) could become difficult or impossible to obtain, risking an inability to operate the software.
- User acceptance for the software might wane, and the user base might erode, as users flock to other vendors with more modern approaches, like updated GUIs, or solutions running on new systems. For example, end users might choose to use windows-based GUIs over their command-line-based ancestors.
- If the software components, languages, or frameworks are becoming obsolete, it might get more difficult and/or costlier to find the required programming expertise (witness the numerous COBOL systems still running in insurance and banking). Maintenance efforts and costs will likely increase over time.

Migration Benefits

- Modern languages and related tools, a larger programmer base, faster hardware, etc., can reduce costs of new feature development, maintenance, and error fixing.
- Modern new software frameworks and libraries can improve the user experience, maintainability, and testability of the system.
- Better APIs can increase interoperability with modern software.
- New platforms (mobile, web, . . .) can open up new markets and increase user acceptance.
- New code can be made more modular using object oriented design patterns, increasing its re-usability, and introduce automated tests (unit tests, integration tests, . . .).
- Vendor and platform dependency can be reduced (e.g., by removing libraries).

Migration Risks

- Obviously, setting up or re-writing software is expensive and the costs are often difficult to estimate. The original software's long-developed optimizations and workarounds might not always be easy to reproduce with completely new technology.
- Choosing new environments, setups, and languages as migration target carries the risk of selecting wrong candidates, like soon-to-be obsolete OSes or language paradigms. New, buzz-word-rich platforms often fade and disappear quite unceremoniously.
- There are considerable risks of introducing bugs or unwanted software behavior. Even seemingly useful bug fixes can lead to problems, e.g., if other systems, aware of the known bug, already compensate for it.

- If parts of the system are not migrated, or if the old software needs to be kept alive (e.g., due to contractual obligations), duplicate code bases need to be maintained, and changes propagated to both.
- If the old system is not documented properly, knowledge that exists only implicitly within the program logic can get lost.
- Domain experts and the developers of the legacy system are probably not available anymore, therefore it can be hard to understand and re-implement the software correctly.

2.7 Soft Migration

Tools and frameworks can greatly facilitate software migrations which results in a less risky process, hereinafter referred to as "soft" migrations. The next subsection gives a general overview on the variety of such migration assistance.

Soft Migration Overview

System Virtual Machines

System VMs (also called Full Virtualization VMs) virtualize the complete operating system to emulate the underlying architecture required by a program. Examples are VirtualBox or VMWare.

Application Virtual Machines

Application VMs (also called Process VMs) run as a normal application inside an existing operating system. They abstract away (most) platform and operating system differences, and therefore allow the creation of platform-independent programs that can be executed using this VM. Examples are the Java Virtual Machine, the Android Runtime (ART), or the Common Language Runtime (used by the .NET Framework).

Integrated Virtual Machines

Integrated VMs can be seen as a subtype of Application VMs, because they are integrated and run within another program (e.g., as a plugin). One popular example are Java Applets, where the JVM is either part of a browser, or added as a plugin. Java Applets are not common anymore, because browsers started to remove the support for such plugins for security reasons [9].

Transpilers

A transpiler is a source-to-source compiler. It compiles or translates one language to another and therefore enables code reuse between different programming languages. Examples are GWT (see section 3.4), which transpiles from Java to JavaScript, or J2ObjC[1], which transpiles from Java to Objective-C.

[1]http://j2objc.org/

Delegates/Wrappers

Delegates or wrappers are tools that allow interaction between system and programming language boundaries. There are several reasons to create a wrapper (like security, or usage of a different programming language), but the basic idea is to hide the underlying program and instead provide a suitable interface for the user. Examples are libraries that allow to call from COBOL to Java[2], or from Java to .NET.[3]

Distribution Utilities/Platforms

These are tools to facilitate the installing and updating of applications. One example is Java Web Start, which is basically a protocol for a standardized way to distribute Java applications and their updates. Other examples are digital distribution platforms like Google Play Store or the Apple App Store. These platforms also help with the distribution of an application, but in addition provide ways to search for and advertize programs.

Java-Based Soft Migration

This section describes soft-migration approaches in the context of the Java platform in more detail. Java has several properties that make it a good example for software migration: it is designed for platform independence, which facilitates, e.g., mere migrations to new operating systems; it is very popular and thus there exist a wide variety of support tools; and several of its language features make concurrent support of different platforms easier than with other programming languages.

Idea

As mentioned, programs written in Java can be run on many different platforms, such as Windows, Linux, OSX or Android, where the code is executed by a Virtual Machine (e.g., JVM or Dalvik). In addition, there are other platforms such as iOS or the web, where the code must be transpiled to another language to be runnable (e.g., GWT can be used to transpile from Java to JavaScript which runs in a web browser). Unfortunately, not the full Java API is available on all such platforms—therefore core Java code that is to be run on various platforms needs to be more restrictive in terms of API usage than the rest of the code.

The idea of reusing program logic on several platforms and programs, even if they do not use the same programming language, is not new. Most client/server applications already hide their internally used programming language(s) by providing a standardized type of API (e.g., CORBA, JAX-WS, or REST). This enables several programs to reuse certain functionality as if it were part of their own application code. The idea of reusing program logic on several platforms and programs, even if they do not use the same programming language, is not new. Most client/server applications already hide their internally used programming language(s) by providing a standardized type of API (e.g., CORBA, JAX-WS, or REST). This enables several programs to reuse certain functionality as if it were part of their own application code.

[2] `http://supportline.microfocus.com/documentation/books/nx40/dijint.htm`
[3] `http://www.ikvm.net/`

This soft-migration approach also encapsulates the shared functionality behind a specific API and allows different programs to reuse it. If these programs use different programming languages, the language barrier can be avoided by using transpilers (e.g., to JavaScript with GWT, or to Objective-C with J2Objc).

Steps

The steps are inspired by the book **Software Evolution** [61] and the **horseshoe process model for reengineering** which can be found in the books chapter 1.2.2.

1. **Analyze the current application.** The first goal must be to understand the legacy system in its current form. The core concepts must be abstracted and a high-level architectural model must be created. Ideally, the system is amply documented; in practice, some reverse-engineering is often inevitable.

2. **Improve the architectural model.** To support migration, or to extract reusable core components, the high-level architectural model usually must be improved. This typically leads to improved modularization of the application and to the creation of a clearer, layered architectural model.

3. **Re-engineer the application.** The next step is the implementation of the improved model. This is also typically the most complex step. Special care must be taken not to break original functionality, e.g., via—possibly newly introduced—unit tests. Documentation must be updated and/or kept in sync with the changes. Organically grown extensions and ad-hoc solutions or fixes should be ironed out. This is also an opportunity to clean up naming conventions, as well as build processes.

4. **Migrate to the new platform.** After the necessary reengineering steps are completed, the new platform specific code must be implemented. If the previous steps were successfully implemented, there should be clear interfaces to the shared codebase.

5. **Optional: remove code for old platform.** If the old platform should be dropped, its platform-specific code can be removed. This helps minimize maintenance efforts—even "dead" code causes obstacles when browsing/understanding a code base.

Supporting Technology

The Java platform offers many tools that help in keeping the codebase maintainable and modular. The following list presents some important categories of tools, and lists some examples.

- **Automated Tests.** Typically, legacy code-bases have no automated tests, therefore it is risky to refactor such code, because any change can easily break previously working features. Therefore it's usually a good idea to write some tests before refactoring the code. A useful tool to write and execute tests for Java code is JUnit.[4] It can be combined with

[4]`http://junit.org/`

Mockito[5] to create simple mocks of dependencies. Combined with Powermock[6], even static fields, final classes, and private methods can be mocked for tests.

As legacy code-bases often consist of tightly coupled components, it might be necessary to break those dependencies (see section 2.7) before writing tests (e.g., a tightly coupled database connection is typically a problem for tests, but a tightly coupled utility class might not). Unfortunately, breaking those dependencies also involves code changes. Feathers [32] describes this vicious cycle of avoiding bugs by making code testable through changes that can potentially introduce new bugs.

- **Dependency Injection.** This implements the principle of **Inversion of Control** for resolving the dependencies of a class. It basically means that objects do not instantiate their dependencies themselves, but get them injected either manually using the constructor, or by a dependency injection framework. Martin Fowler [34] describes the pattern in detail and compares it to some alternatives (like the Service Locator pattern).

  The advantages of using dependency injection become apparent in this migration-approach, as the shared code must not depend on the platform-specific implementation of any dependency. Instead, it should only depend on a platform-independent interface, which, in turn, has one implementation for each platform. Examples for frameworks supporting dependency injection are Google Guice[7] or Spring.[8]

- **Static Code Analysis Tools.** These tools can find potential bugs, dead or duplicate code, and they can help to enforce a common code style. Examples: FindBugs,[9] PMD,[10] or Checkstyle.[11]

- **Build and Dependency Management.** Tools like Apache Maven[12] or Gradle[13] manage the dependencies of an application and its submodules. They also standardize several other aspects of an application, such as the directory structure and the build process. Their "convention over configuration" [20] approach also helps to familiarize new developers with the code-base, because of familiar project structure conventions.

[5] `http://code.google.com/p/mockito/`
[6] `http://www.powermock.org/`
[7] `http://github.com/google/guice`
[8] `http://spring.io/`
[9] `http://findbugs.sourceforge.net/`
[10] `http://pmd.sourceforge.net/`
[11] `http://checkstyle.sourceforge.net/`
[12] `http://maven.apache.org`
[13] `http://www.gradle.org/`

CHAPTER 3

State of the Art

Before describing details of the migration, this chapter discusses the state of the art of UMLet's technical environment and area of application.

The technical background of UMLet's migration to the web is focusing on software portability in general with a spotlight on the Java programming language. It continues with the current state of web applications and related technologies such as HTML5 and JavaScript. Special attention is given to GWT as a tool to bridge the gap between the Java and JavaScript world.

The area of application introduces the Unified Modeling Language (UML) and UMLet, which is the software to be migrated. It concludes with a comparison of existing web UML tools, which may be used for purposes of inspiration and orientation for UMLet's web version.

3.1 Portability

The history of programming languages shows, that they typically evolved from low-level, hardware-dependent languages to more abstract, high-level languages. With this evolution, code got more and more portable and less dependent on a specific platform.

Machine Code and Assembly Code

In the early days of computers, there were many processors with many different machine or assembly languages.

Machine code is just binary code that can be executed directly by the CPU. Even though Assembly Code is a simple abstraction for the programmer, because the instructions consist of plain-text and human readable commands (like MULT for multiplication) instead of 1's and 0's, but it provides no benefit in terms of portability as there is a strong (often one to one) correspondence of assembly code and machine code instructions.

Therefore applications had to be written for exactly one computing environment. There is practically no possibility of code reuse.

Languages that Compile to Machine Code (e.g., C)

The C Programming Language originated as the language of the UNIX operating system and was a huge step forward, as it is independent of any particular machine architecture, but still matches the capabilities of many computers. One can write portable programs which can be run without changes on a variety of hardware [51].

Although in theory this is possible, it is typically only valid for simple programs, as more complex programs use different APIs on different operation systems and therefore must contain OS-specific code.

C also provides typical control-flow constructions like if-else, switch, while, do, for, break and is therefore much more readable than pure Assembler code.

C compilers are typically very good at optimizing the code during compilation and every C program can use new assembler instructions as soon as they become available and the compiler supports them.

Languages that run on Virtual Machines (e.g., Java)

For portability, this was a huge step forward, as programs are compiled into an intermediate language which is runnable without modifications on any computer which offers an implementation of the required interpreter.

One example for such a language is Java which is described in detail in Section 3.2.

Web Applications

Web applications can be seen as the next step in terms of software portability. Such applications use a web browser as an intermediate layer to avoid dealing directly with the OS. In that sense, a browser behaves similar to a virtual machine, which interprets JavaScript code at runtime, but it provides additional benefits for applications, such as discoverability, easy distribution of updates and a huge user base.

Web applications still have some limitations, such as low performance and limited file and hardware access (see chapter 3.3), but for many end user applications, the advantages like the better discoverability in the open web and the possibility of running on mobile devices (where web browsers are available, but most virtual machines aren't) outweigh the limitations.

On the other hand, certain types of applications like file- and batch-processing software will typically still be implemented as desktop applications.

Operation System (OS) Virtualization and Containers

An alternative approach to portability is OS virtualization. It does not really compete with the other categories, as most of them can be applied on top of a virtualization layer.

The virtualization approach can be divided into **whole-system virtualizers** like VMware[1] or VirtualBox[2] which run the whole Operation System on a virtual machine and **Containers** which are run in an isolated part of an existing Operation System.

[1] `http://www.vmware.com/de`

[2] `https://www.virtualbox.org/`

Both approaches have the advantage that a whole system (e.g., a composition of several applications which should work together in a certain pre-configured way) and its configuration can be shared as one module.

Another advantage over Virtual Machines like the one from Java is that the programmer can use all OS specific functions and tools instead of being limited to the ones which are offered by the VM.

The advantage of **Containers** over **whole-system virtualizers** is that they use normal system calls offered by the local OS and therefore typically have better performance [33].

One of the newer Container based projects which gained much attention is Docker [3]. According to Jay Lyman, a senior analyst for enterprise software at 451 Research, *"Docker is a tool that can package an application and its dependencies in a virtual container that can run on any Linux server."* [2]

Even though it's limited to Linux OS, it is already integrated in many cloud computing platforms like Google Cloud Platform, Microsoft Azure, Amazon Web Services.

Longevity of Programming Languages

As shown, the portability of a program is often linked to its programming language and environment for which it was initially developed.

There are several language rankings [1, 68, 74] which try to compare the popularity of programming languages. The following languages are in the top 10 of all rankings: Java (1995), C (1972), C++ (1983), C# (2001), PHP (1995), JavaScript (1995). These statistics show, that the most popular programming languages are relatively stable.

The reasons for this stability are manifold. Widely used programs with large code bases are written in these languages, there is a huge knowledge base from developers, and the tooling and ecosystem build around these languages is substantial.

In addition to these popular languages, new languages will probably raise in popularity too, while the existing ones will stay relevant for a long time, which means developers will have to deal with many different programming languages. As it can be hard to port an application from one platform to another, the choice of the programming language is very important, because every one of them has different features and restrictions. The right language choice can improve the portability of applications, especially if there exist source-to-source compilers from one language to another.

3.2 Java

Java first appeared in 1995 and is an object-oriented computer programming language. Today it is one of the most popular programming languages according to several programming language popularity ratings [1, 68, 74]. It is mostly known for server-side applications, but it can also be used to write platform independent desktop applications.

[3] `https://www.docker.com/`

Java Platforms

Java can be run on different platforms with different requirements. A **Java Platform**, sometimes also referred as **Java Editions** is specific for a **Java Virtual Machine** (Java's code execution engine) and a certain **Java API**.

The following sections will briefly describe the most important official platforms and their typical use cases and properties. There are also some variants of these platforms, like ME Embeddable, Java TV (based on Java ME), SE Embeddable and others.

Java Card

The target platform are smart cards and other devices with limited memory and processing power. The main goal is to provide a secure environment for such devices [70].

Micro Edition (ME)

The Micro edition targets embedded and mobile systems like mobile phones, micro-controllers, printers, ...

The API is a subset of the Java SE API with several additions which are specific for mobile devices.

Standard Edition (SE)

The Standard Edition targets desktop and server environments.

The reference implementation is called OpenJDK and its source code is licensed under the GPL with linking exception.

Enterprise Edition (EE)

The Enterprise Edition targets multi-tier client-server applications and extends the Standard Edition with several APIs for use cases like:

- **Java Server Faces (JSF)** for building user interfaces for web applications.
- **Contexts and Dependency Injection (CDI)** to support implementation of the **Dependency Injection** design pattern.
- **Enterprise Java Beans (EJB)** to build modular components of enterprise applications.
- **Java Persistence API (JPA)** as an object relational mapping layer to relational databases.
- **Java API for XML/RESTful Web Services (JAX-WS/JAX-RS)** as an API which helps creating REST or SOAP based web services.

Java Virtual Machine (JVM)

Java programs are typically compiled to Java byte-code (.class files) which is then executed on a **Java Virtual Machine (JVM)**.

The advantage of this indirection is also the basis of the Java slogan **write once, run anywhere**, which means the java byte-code runs on any device and operation system for which a JVM has been implemented.

The idea of such a virtual machine was very appealing for many developers, therefore as of today, there exist many different JVM implementations [67] and programming languages which are compiled to Java byte-code such as Jython[4] and Scala[5].

Unofficial (not JVM based) Platforms

Because of the popularity of the Java programming language and its ecosystem with tools such as IDEs (Eclipse, Netbeans, ...), continuous integration systems (like Jenkins), build management tools (like Maven) and much more, other companies reused Java for their own purposes.

2 of the most important examples from Google are:

- **Android OS**: Most applications for the Android OS are written in Java using a limited set of the Java SE API (e.g., without AWT and Swing classes), but the code is compiled to dalvik byte-code and run on a Dalvik VM.

- **Google Web Toolkit (GWT)**: an open source toolkit to develop web applications (see chapter 3.4).

3.3 Web Applications

The web is constantly evolving and web applications are a popular way to benefit from the discoverability and platform independence of the world wide wide.

Tools for the World Wide Web

The following listing presents the main tools for the web based on their approximate historical appearance. Web browsers evolved from simple document viewers to application platforms with a wide spectrum of functions, comparable to native operating systems.

Web Browser

The web browser is used to retrieve and present information from the world wide web. The early days of the web were all about serving static web pages or documents from a server to clients. The pages were written in HyperText Markup Language (HTML) and styled using Cascading Style Sheets (CSS). Therefore the browser was basically just a document viewer, and the only way of limited interactive user input was provided by HTML forms.

[4]Jython is a Java based implementation of the Python programming language

[5]Scala is a functional/object-oriented programming language as a concise alternative to Java

HTTP

The Hypertext Transfer Protocol (HTTP) is a stateless protocol for data communication in the web. The statelessness is sufficient for serving documents, but many web applications need a way to store the client state persistently. Even a simple feature, such as remembering the clients log-in on a web page, needs such a storage.

Cookies

Cookies provide a way to store the web application state on the client computer, essentially solving the issue of persisting the client state. This feature also enables web applications to tailor a custom document to the specific user instead of serving the same document to everybody.

The introduction of cookies were an important step from stateless document serving applications to primitive web applications.

JavaScript

JavaScript is a dynamic computer programming language which is (mostly) run within the clients browser. At first it was used for small alterations of a web page without requests to the server, but the real potential of JavaScript was recognized with the advent of AJAX, which is the principle of sending asynchronous requests to the server to replace only small portions of the web page, instead of retrieving the whole document. This allowed web applications to appear much more fluid and “application like“ to the user.

Later the standardization of JavaScript (as ECMAScript) helped to provide the same program experience over all browsers and the browsers made huge improvements to the JavaScript performance, which made it possible to create complex web applications which behave mostly like classic client applications.

Browser Plugins

There exist alternative software platforms to write applications, such as **Java Applets**, **Microsoft Silverlight** or **Adobe Flash** which can be embedded into the browser via plugins.

Historically they are interesting because they enabled web developers to build interactive web applications at times, when JavaScript was too slow for this purpose.

In the modern web, the plugin based architecture is problematic, due to security and portability issues. For example, Java Applets are not available on mobile browsers and some desktop browsers already stopped supporting them, such as Chrome on September 1, 2015 [9], Flash is not available on iOS and newer Android devices, Silverlight is only available on Windows and Mac OS X, but Microsoft plans to end the support in 2021 [6]

[6]Microsoft Support Lifecycle `https://support.microsoft.com/lifecycle/?LN=en-us&c2=12905`

HTML 5

The Hypertext Markup Language (HTML) is the standard markup language for web sites. HTML 5 introduced many new HTML Tags like <video>, <audio>, <canvas>, which help to express the semantics of web pages. These new features are typical building blocks of modern web applications (e.g., a game will draw it's content on a canvas, a video player can show mark the video area with the appropriate tag and so on)

There are also many HTML 5 related specifications (with different state of development) which help in building complex web applications like:

- **Web Storage**[7] an improvement over cookies in terms of size, usability and API.
- **File API**[8]**:** an API to access and read files, write files and access client file systems.
- **Web Socket API**[9]**:** allows full-duplex TCP connections between clients.
- **Clipboard API**[10]**:** provides APIs for clipboard operations like copy, cut, paste.
- **Web Workers**[11]**:** allows web developers to use background workers for thread-like programming.

Restrictions Compared to Classic Standalone Applications

Although many restrictions of web applications have been loosened with improvements to browsers and web standards like HTML 5 and JavaScript, there are still some differences which must be considered as a web developer.

HTML 5 Support in Browsers

Although many features of typical desktop applications are already contained in the HTML 5 (and related) W3C recommendations, the support in different browsers must be checked before developing a web application[12].

There are websites like `http://caniuse.com/` which list the current state of development for each feature and browser.

Restrictions of HTML 5 Features

Even if a feature is implemented using HTML 5, there are often additional restrictions which are necessary due to security requirements and the nature of web applications (native applications must be actively installed, while web applications work by accessing an URL).

[7]Web Storage W3C link: `http://dev.w3.org/html5/webstorage/`:

[8]File API related W3C links: `http://www.w3.org/TR/FileAPI`, `http://www.w3.org/TR/file-writer-api`, `http://dev.w3.org/2009/dap/file-system/file-dir-sys.html`

[9]Web Socket API W3C link: `http://dev.w3.org/html5/websockets/`

[10]Clipboard API and events W3C link:`http://www.w3.org/TR/clipboard-apis/`

[11]Web Workers W3C link:`http://www.w3.org/TR/workers/`

[12]Some websites offer such HTML 5 support tests like `http://html5test.com/`

For example, the FileWriter API is a feature which can be abused easily if it would work like in native applications (by allowing access to any file the OS provides access to). Therefore the browser support is very low (only Chrome and Opera support it [66]) and you will typically have to think about alternative ways to store information (e.g., by using the Web Storage API instead of writing into files on the filesystem).

Performance Restrictions

Although browsers JavaScript execution performance has improved drastically over the last years, they typically do not exceed native code performance in general.

There are different efforts by browser vendors (like asm.js from Firefox or the JavaScript alternative Dart from Google) but there is no consensus about the best way to improve client side programming performance in the future.

Another performance issue is that JavaScript basically runs sequential and does not offer parallel execution (as multi-threaded native code would allow). However this issue is alleviated with the introduction of the Web Workers API, although they are relatively heavy-weight and should not be used in large numbers according to the WHATWG [48].

Limited Hardware Access

Currently web applications are limited in terms of hardware interaction (like a Microphone, Camera, Bluetooth devices, Accelerometer, ...) Currently most of these things cannot be accessed using a standardized API, although there are some efforts and browser-specific ways to access some of them [49].

Even if there are such APIs in the future, they will likely be less flexible and powerful than native counterparts, because of Security related restrictions.

Differences in Browser Implementations

Historically there were huge differences in HTML, CSS and JavaScript interpretations between different browsers. Although there have been huge improvements over the last decade, web applications still have to be tested on all of the target browsers to make sure they work as expected.

For public web applications, monitoring the browser distribution is also very important, to make sure the targeted user group is captured. E.g., according to NetMarketShare [3], Internet Explorer 8 and 9 are still used by nearly 10% of the people, and these browsers typically do not support most of the newer HTML 5 features.

Mobile Web Applications

Traditionally web pages have been accessed almost exclusively from desktop computers or laptops. However, in the last years a large portion of web traffic has shifted to mobile, touch based devices.

Therefore web applications do not only have to consider classic computers (mouse-controls; large horizontal display; fast internet connection), but also mobile devices (touch-controls; small vertical display; slow internet connection).

Controls

There are many differences between mouse and touch based controls. A mouse typically offers a second button to open the context menu, which is not possible using a simple touch. On the other hand touch-controls can be used with more than only one finger, and therefore offer new possible ways of interaction.

Another important difference is the precision. A mouse click targets exactly one pixel, but a touch covers a larger area. Combined with the smaller display size of mobile devices, interactive elements like buttons typically have to be much larger than before, but the larger buttons can feel unnatural if the user uses a desktop computer with a mouse.

Display

In addition to the mentioned problems due to the different display size and controls, the screen alignment is typically horizontal on desktop computers and vertical on mobile devices.

Therefore the positioning of web application elements should adapt to the type of device it is used on (e.g., the desktop version of the web application should place a menu on the left side of the screen, the mobile version should place it on top)

Because of these differences, applications which are used on both types of devices should typically offer customizations, e.g., by using different CSS style sheets for each environment.

Internet Connection

Web developers should consider the rather unpredictable and slower internet connection when thinking about which parts of the application should be handled on the client side and which ones on the server side and how and when resources are retrieved from the server.

E.g., if the application grows large, it is typically a good idea to split it into several parts which are retrieved when used instead of all at once at the beginning.

Calculations and changes to the web application which do not need to be done on the server side should probably be moved to the client side, although very complex calculations which cannot be done on usually slower mobile CPUs should remain on the server side.

Conclusion

The differences between classic desktop devices and mobile devices are manifold and should be considered during application development.

However the amount of time which must be invested for such customization depends on the typical use case of the application.

A classic web application which is only presenting documents to the user may only need simple adaptions using a specific CSS style sheet.

A full-fledged web application like a computer game typically needs specific customizations based on device type, controls, screen resolution, and more.

3.4 Google Web Toolkit (GWT)

GWT has been developed by Google which has released version 1.0 on 2006-05-17. Today it is maintained by the **GWT Steering committee** and is an open source toolkit to develop JavaScript applications by writing Java code and compiling it to JavaScript.

It generates different JavaScript files for different browsers to support most common types of browser quirks (although today most browsers interpret JavaScript in a similar way, this was an important feature in the early days of GWT).

It emulates many important parts of the Java SE API [29] and also offers 2 different developer modes for easy debugging and fast compile-cycles during development.

- **(Classic) Development Mode**: The client side Java code is not compiled to JavaScript, but executed as compiled Java byte-code in the JVM of a code server. The browser uses a plugin to communicate with the code server.

- **Super Dev Mode**: As maintaining browser plugins can be tedious, this mode offers source map based debugging using the browsers developer tools. In this case the client code is compiled to JavaScript.

Additional Features

In addition to the main features which are about Java to JavaScript compilation, GWT has also been extended over the years with many different useful tools for Web UI development, such as

- **UI Widget library** to create websites which look and behave the same accross all browsers.

- **RPC framework** as a lightweight method to transfer data between client and server side.

- **Browser History Management** by either using a simple URL-Parameter API for manual History handling or using the full-fledged **Activity & Places framework** which offers URL-centric website-navigation and life-cycle handling of views and their corresponding application-logic (Activities) [13].

- **Support for Unit Tests** by either extending **GWTTestCase** which emulates the application behavior by launching it in a HtmlUnit browser or by mocking UI classes using the **GWTMockUtilities**.

- **UiBinder** which allows to develop static parts of the app as HTML pages with CSS and only the dynamic parts with Java-Code.

- Support for **local CSS and Resources** such as images and text files for scoped widgets with encapsulated styling and resources.

- **Internationalization** of applications by simple translation into different languages.

[13]further information about the Activity & Places framework framework can be found at `https://ronanquillevere.github.io/2013/03/03/activities-places-intro.html`

- **Security** features such as Cross-Site-Scripting and Cross-Site Request Forgery protection.
- **Accessibility** support for screen readers.
- **Logging** support for the development mode, browser console, firebug or pop-ups.

The documentation for all of these features is comprehensive and can be found at the GWT website [26].

Alternatives to GWT

There are some alternatives to run Java code in a browser without using a plugin.

- **Doppio** and **Node-jvm** are Java Virtual Machines written in JavaScript. These tools may be appropriate for use cases where existing jar files should be run on machines without a JVM. They don't fit the use case of this migration, because no user would wait until the whole JVM and class library is downloaded. Also the performance is most probably worse than with a transpiler.
- **TeaVM** and **Dragome** transpile Java byte code (instead of source code) to JavaScript. They do not offer the extensive Java API emulation or previously mentioned additional features of GWT, but they can also transpile languages, which output Java bytecode (such as Scala, Kotlin, ...)

Although some of these tools may be the better choice for other use cases, the migration of UMLet uses GWT for the following reasons:

- Extensive Java API Emulation (especially important for migration of existing software, where it's hard to avoid usage of specific parts of the API)
- Large user-base, many libraries, support by major companies (Google, Vaadin, ...)
- Actively developed (GWT 2.8 introduced support for most of the new Java 8 API and syntax such as Lambda functions)
- Highly optimized output, therefore fast and small applications which is important for GUI-centric interactive web apps

3.5 Unified Modeling Language (UML)

UML is a ISO/IEC standardized general-purpose modeling language, used in software engineering [83, 93]. It is used to create visual models of the behavior and structure of object-oriented software systems, and helps during the specification, construction and documentation of such systems.

History

Jim Rumbaugh, Grady Booch and Ivar Jacobson developed the first version from 1994 until 1997. After they finished their work and released it as Version 1.1, it has been standardized by the **Object Management Group (OGM)** [41] in 1997. After several minor revisions (1.3, 1.4, 1.5), a new major revision was released as Version 2.0 in 2005. As of today, the most current version is v2.4.1 which has been released in August 2011.

Diagrams

UML divides its diagrams into two categories: structural diagrams and behavioral diagrams.

Structural diagrams are mostly used to document the software architecture, while behavioral diagrams describe the behavior, functionality and interactions within software systems [69].

An overview of the diagram types and their hierarchical order and relations is shown in 3.1

3.6 UMLet

UMLet is an open source tool for quick and easy creation of different types of diagrams, developed and maintained at the Vienna University of Technology since 2001. It is available as a standalone Java Swing desktop application and as plugin for the Eclipse IDE. Both versions are widely used and well known to many students and professional software developers.

UMLet is one of the most prominent open-source UML tools today with 730,000 page views of www.umlet.com in 2015 from 203 countries (according to Google Analytics). It is the most favored plugin on Eclipse Marketplace [31], the Xmarks page ranking lists `http://www.umlet.com` at rank 2 for the category ”Uml Tools” and rank 7 for the category ”Uml” [91]. UMLet has been mentioned in various books [53, 84, 86] and has been the topic of several theses [37, 63, 73] and conferences [4–7].

User Interface

UMLet shows the current diagram at the left side of the program, while the right side is split into the upper part which shows a palette of elements and the lower part which shows the properties of the selected element (see chapter 3.2).

It offers different element palettes logically grouped together (e.g., UML Class diagram palette, UML Use Case diagram palette, . . .) from which the user can add elements to the diagram via double-click, drag&drop or copy&paste. Positioning of diagram elements is done by moving elements around with the mouse or keyboard.

Diagram Sharing

Diagrams can be saved as UXF files, which is a XML based representation of a UMLet diagram, or exported to the following formats: BMP, GIF, JPG, PNG, PDF, SVG, EPS.

Alternatively diagrams can also be shared by sending them as email-attachment using a simple E-Mail interface embedded in UMLet.

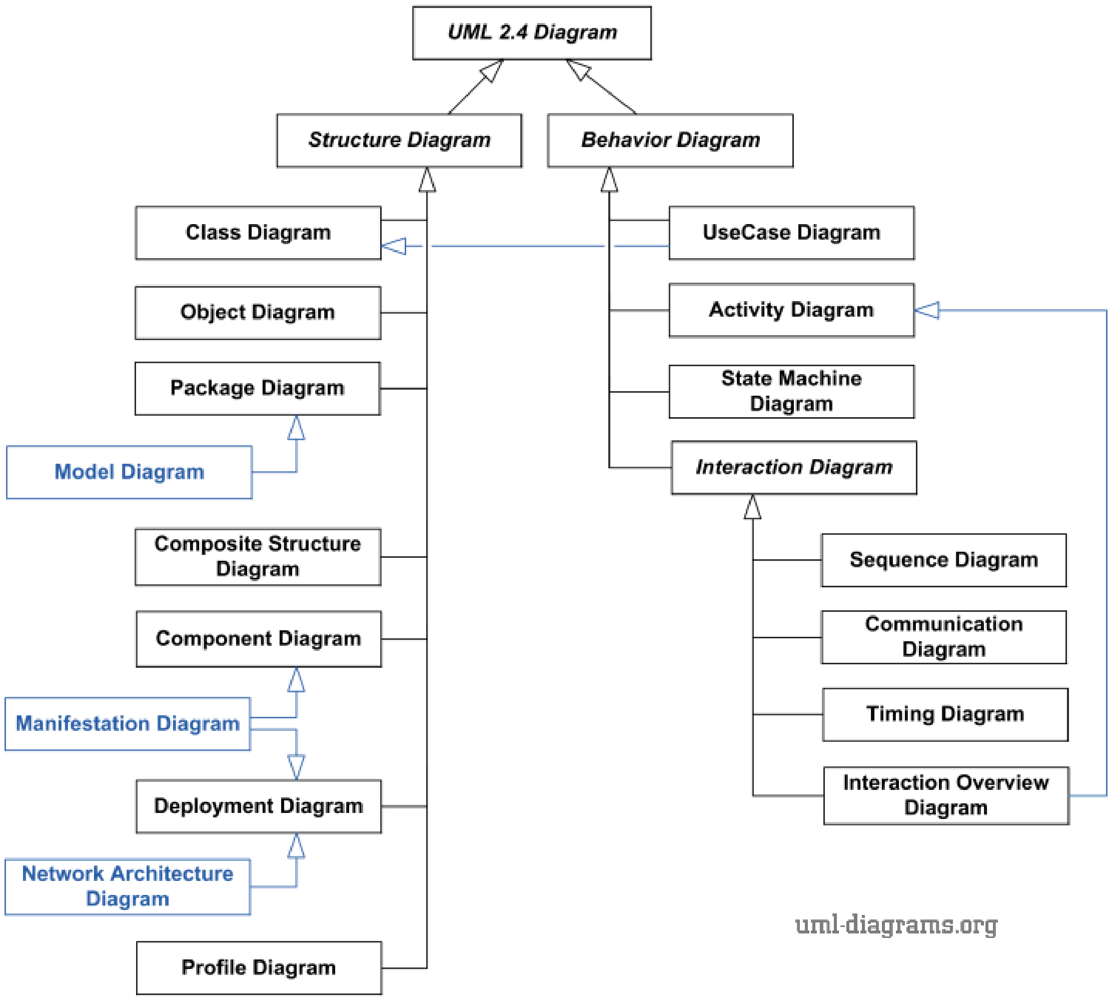


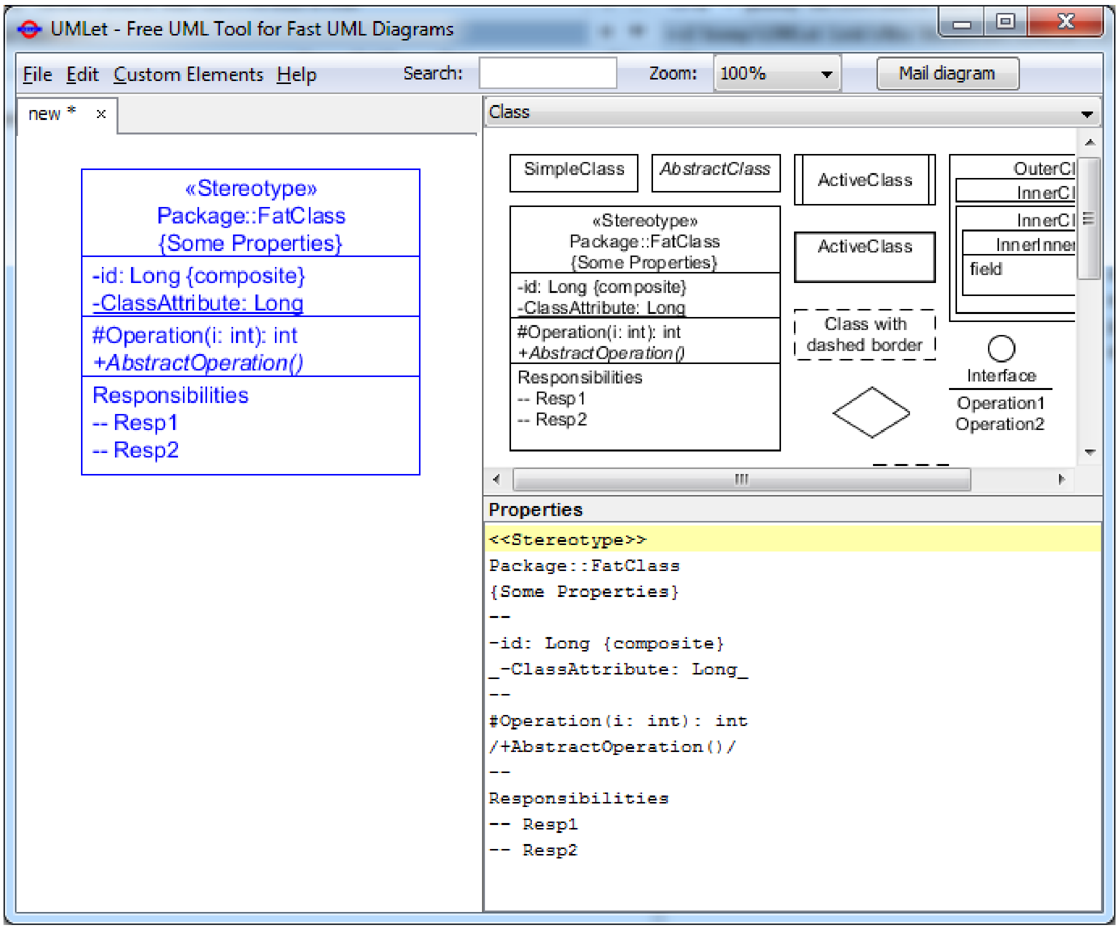
Figure 3.1: UML 2.4 Diagrams Overview (Source: www.uml-diagrams.org [69])

Diagram Elements (GridElements)

Most elements which UMLet offers are from the UML- Specification [83, 93], but it is also possible to create other elements (e.g., for ER-Diagrams or self-defined custom elements). All elements can be categorized as follows.

Simple Element

These elements represent a small portion of a diagram (e.g., one Class or one Use Case) and can be customized with predefined functions, such as **fg=red** to set the foreground color to red. They represent the majority of UMLet elements.



```
«Stereotype»
Package::FatClass
{Some Properties}
-id: Long {composite}
-ClassAttribute: Long
#Operation(i: int): int
+AbstractOperation()
Responsibilities
-- Resp1
-- Resp2
```

```
<<Stereotype>>
Package::FatClass
{Some Properties}
--
-id: Long {composite}
_-ClassAttribute: Long_
--
#Operation(i: int): int
/+AbstractOperation()/
--
Responsibilities
-- Resp1
-- Resp2
```

Figure 3.2: UMLet v12.2 on Windows 7

Custom Element

Custom elements work like Simple Elements, but the user can define in detail what the element should look like. Each Custom Element consists of Java Code (compiled and executed at runtime), which will define where shapes and text should be drawn. The user can use typical Java constructs like if/else and for-loops and use predefined basic draw-methods (like drawRectangle(x,y,width,height)) to specify the exact look of the element.

All In One Diagrams

Although technically this is also a Simple Element, it is typically used as the only element to draw a whole diagram.

Currently only UML Sequence Diagrams and Activity Diagrams can are available. Both of them use a distinct syntax to create a textual representation of a whole diagram which is then parsed and converted to a visual diagram.

Diagram Element Customization

Customization of elements is done by using predefined functions which are put directly into the properties text. Every line which is not a function is simply placed into the element as text.

Some of the customizations (like setting the foreground color or setting the background color) will work for many different elements, while others (like the template class modification) are restricted to some specific elements.

For example, the following properties text contains customizations to the foreground and background color, a horizontal line and a template class text. If the text is put in a class element, it will look like 3.3, if it is put in a use case element, it will look like 3.4.

```
template=Element: Object
ArrayList
--
+add(e: Element)
fg=blue
bg=yellow
```

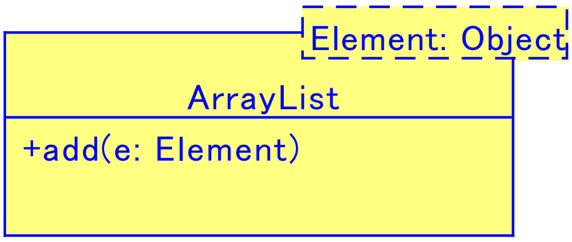


Figure 3.3: simple Template Class with foreground and background color set

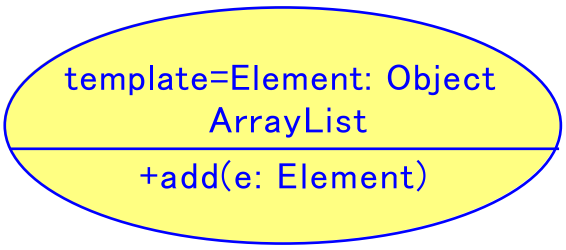


Figure 3.4: Simple use case with fgcolor and bgcolor set

As it can be seen, the foreground and background color works as expected for both elements, but the template function is only interpreted as a function in the class element. The use case element doesn't know it and therefore prints it as text.

The example also shows that elements will resolve some functions, such as the **--**, slightly different. While both elements draw a horizontal line, the class element also changes the horizontal text alignment after the line from center-aligned to left-aligned.

Platform Evolution

UMLet started out as v0.1 on 2001-01-25 as a Java Applet with a server side diagram repository based on JSP technology. Diagrams were stored in a database with a specific schema instead of XML files. At the time it was a modern approach but suffered from problems which are mostly inherent to the Java Applet technology like:

- long start-up time of the Java Virtual Machine
- problems with different versions of installed and required Java Runtime Environments
- unpredictable download size (depending on the Java installation on the client side)
- requires browser plugin
- therefore limited browser and OS support

Because of these problems and the maintenance effort of running the server to store diagrams, the program was changed to a simple jar-file which can be run on any computer where a JVM is available. Diagrams are now stored in XML based UXF files.

Later the development of the Eclipse Plugin version started as an alternative for users of the Eclipse platform.

The goal of this thesis and the accompanying reengineering of UMLet is to bring the program back to web-browsers as an additional platform, but without the mentioned problems of Java Applets.

The resulting program is called UMLetino and it will be the first UMLet version which does not require an installed Java Runtime Environment and therefore will reach several new platforms like the new mobile operation systems (e.g., Android, iOS) or environments where the user does not have the right to install software.

3.7 Web Based UML Tools

There are many different tools to generate UML diagrams. Some of them are more like lightweight tools to create quick UML sketches (like UMLet), and others offer a complete integration of UML with an IDE (Integrated development environment) and code to diagram, respectively diagram to code generation (e.g., Rational Software Architect from IBM).

This section focuses on existing lightweight web based UML tools, as this is the main goal of the GWT based UMLet variant which will be discussed in this diploma thesis.

yUML

- Website: `http://yuml.me/diagram/scruffy/class/draw`

Offers simple creation of class diagrams, activity diagrams and usecase diagrams and is completely free to use. There is also a business license which includes private diagrams, private hosting, source code and support.

The syntax is similar to UMLet's All-in-one diagrams, which means the user doesn't drag diagram parts around manually, or specify every single diagram element and it's content, but instead describes the whole diagram in a specific text based syntax from which the tool will then create the elements and relations between the elements.

Also it provides different stylings for the diagrams (e.g., one style is plain and simple, another one imitated hand written diagrams)

sketchboard

- Website: `http://sketchboard.me`

Can create most types of UML diagrams (and custom own images for business account users). Is free for public projects, and offers business licenses for private usage.

Interaction with elements is based on drag and drop. Similar to UMLet's non-all-in-one elements the user drags elements from a palette into the diagram and specifies its properties.

Further customizations (like background color) can be made using a pop-up panel over the element which appears after selection.

To create a relation to another element, the user has to select an element which lets 4 drag-points appear (one for each direction: left, right, upper and lower). Then he has to drag a relation out of the element by holding the mouse button and release it where the new target-element is located. If there is no target-element, a pop-up appears and allows to create a new element.

acsiiflow

- Website: `http://www.asciiflow.com`

This tool is free to use and works like a simple graphics painting program, but its content is solely based on text characters (mostly ASCII, but text content can also have special characters).

One the one hand this is limiting the possible customizations of diagram elements, but on the other hand the output is simple ASCII text and is easily exported and imported back into any text file.

The idea to use simple text content to create complex diagrams is somehow similar to UMLet's idea of specifying element properties via text functions.

js-sequence-diagrams

- Website: `http://bramp.github.io/js-sequence-diagrams/`

This tool can create UML sequence diagrams from text written in a specific syntax and is therefore similar to yUML or UMLet’s All-in-one diagrams.

It also offers customization of the diagram by setting the style to “simple“ or “handwritten“.

An interesting aspect is the syntax which is similar to English sentences, e.g., “Note left of A: <text>“ will place <text> to the left of the element A.

websequencediagrams

- Website: `http://www.websequencediagrams.com/`

This is another tool to create sequence diagrams using a syntax which is like the all-in-one diagrams.

One interesting aspect is that there is a palette which shows common parts of a sequence diagram which can be added to the current diagram.

Another nice feature is the link between the selected part of the text and the part of the diagram. If the user selects a line of the text, the relevant part is highlighted in the diagram and if the user selects a part of the diagram, the text line is highlighted.

CHAPTER 4

UMLet Evolution Approach

This chapter applies the software evolution criteria from section 2.6 to UMLet, to find the best evolution approach for UMLet.

Afterwards, the first two steps of a soft migration (see section 2.7) are applied, which means the current application is analyzed and documented by creating a high-level architectural model. This model gets improved in two major ways.

- The old architecture of GridElements was not suitable for shared functions (which are described in section 5.1), and it coupled elements to the Swing API. To solve those problems, the architecture has been redesigned, in order to enable such functions and to remove Swing specific code from the GridElements (in preparation for the web version which draws on an HTML canvas).
- A new web platform specific variant of the model is created, which is based on the existing high-level architectural model, but applies some changes which are required to work for the web.

4.1 Decision Drivers

The main decision drivers for the evolution of UMLet are:

- The current platform (Java virtual machine on top of an OS) is not future-proof:
  - Java often does not come pre-installed and user may not be permitted to install it (e.g., admin rights missing). It is not unlikely that future closed-source OS iterations further discourage Java deployments.
  - New operating systems (Android, iOS, Chrome OS) have no Java virtual machine available

  - OS vendors increasingly limit the installation of unsigned software, or try to coax applications to be provided via custom app stores. This gives vendors the influence to prohibit flexible, uncomplicated installs for casual users, and also allows them to ban applications outright (e.g., if an application does not comply with some user interface guidelines, if the vendor perceives its usability or uniqueness as not adequate, or if tech specs like access right handling are not to the vendor's liking).

- The user base might move to web-based solutions which do not require an installation, e.g., yUML, sketchboard, js-sequence-diagrams, websequencediagrams (see section 3.7).
- The codebase of UMLet got increasingly complex over the last years (20+ contributors without lead designer), therefore the architecture of UMLet should be re-evaluated to make sure it's future proof, and it enables a potential migration to a new platform.

4.2 Evaluation of Software Evolution Criteria

When the software evolution criteria list from section 2.6 is applied to compare the solution of preserving the existing platforms and migrating to the web platform, the following results can be identified:

Preservation Benefits

- The stability of the program is preserved (less risk for new bugs)
- Existing platform support can be improved instead

Preservation Risks

- Code is hard to maintain and platforms are already mixed (Eclipse plugin and standalone)
- Users might switch to other web UML tools
- Maintenance will become more difficult over time

Migration Benefits

- Availability on Android, iOS, ChromeOS
- Modularization of project improves maintainability
- Dependency to Java installation removed

Migration Risks

- Users might not accept the web platform
- Browsers may not be ready for GUI-heavy applications
- Browser runs JavaScript, UMLet is written in Java

4.3 Results of the Evaluation

The decision drivers and the evaluation of the software evolution criteria show that a migration to the web platform should be beneficial for UMLet, even though there is a risk involved that not all features of the program will work within a browser due to the complex GUI and user interaction of UMLet. To mitigate the risks involved with such a migration, the goal is to keep the current platforms, as well as most of the current business logic. This will avoid the risk of users switching to other tools if the web platform doesn't fit their needs.

To bridge the gap between Java (the language in which UMLet is written) and JavaScript (the language of the web), a transpiler such as GWT should be used. GWT transforms the Java codebase into JavaScript, which can be run within a browser without a plugin.

The goal is to separate UMLet's codebase into modules with clear dependencies between each other. All platforms should share as much code as possible, and each platform should have a separate endpoint module which encapsulates the code which is only needed for this platform. In other words, the code should be redesigned in a way which embraces the multi-platform nature of UMLet.

Steps

The steps from chapter 2.7 are applied to the UMLet web-migration which results in the following tasks:

1. **Analyze the legacy application** in its current form by extracting information through analysis and abstract the core concepts to create a high-level architectural model (see chapter 4.4).

2. **Improve the architectural model** by restructuring and redesigning the core components of the program. The goal is clear separation of concerns and a layered architecture (see chapter 4.5).

3. **Reengineer the application** and implement the changes by refactoring and by applying the principle of Separation of Concerns. The most important separation is the split-up into the shared code with clear and simple interfaces and the standalone-specific code which implements them (see chapter 5).

4. **Migrate to the web platform** by implementing the interfaces to the shared codebase, writing additional web-specific code and compiling the first version of UMLetino (see chapter 6).

4.4 Current High-Level Architectural Model

Figure 4.1 shows the High-Level Architecture of UMLet.

The most important part of the model is the **GridElement** which represents every element drawn on the diagram. In the codebase all UMLet elements (e.g., Use Case, Class, Activity,

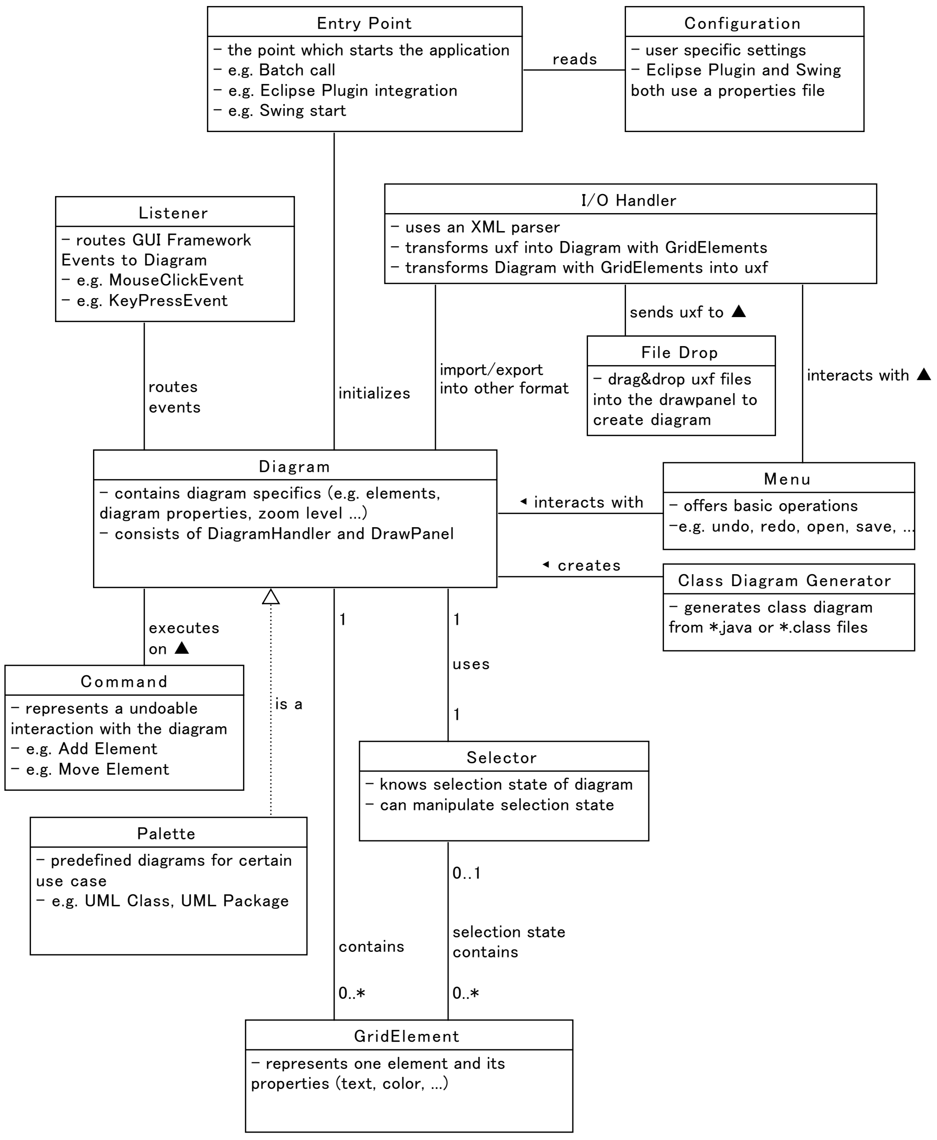
. . .) extend the abstract parent class GridElement. Together these elements represent the main functionality of an UML modeling tool and also make up the majority of UMLet's source-code.

In other words, the GridElements provide the main recognition value of UMLet. If they look and behave the same in all UMLet variations, the user will immediately feel familiar with the program on any platform. The other parts of figure 4.1 are just there to enable and improve the interaction between the user and the diagram which consists of GridElements.

Other Parts of the Model

- **Configuration:** Standalone programs (including UMLet) typically use configuration files, but web applications have only very limited access to the file system 3.3, therefore the web application must use a different approach as described in 10.3.
- **Entry Point:** The standalone application has several different entry points like the batch-mode without showing a Swing GUI, the Swing GUI and the Eclipse Plugin integration.
- **Listeners:** Although the Listener concept is used on most platforms to react to user input from a GUI, the implementation is strongly tied to the specific platform, therefore the listener-specific code is typically not reusable.
- **I/O Handler:** Like the Configuration, I/O on a standalone application is largely file based (importing diagrams stored as uxf and exporting into uxf or different output formats like bmp, jpg, png, svg) and therefore different from the approach a typical web application will use. However some parts like the XML parsing of uxf files can be extracted in a reusable component.
- **Menu:** Although every platform will have some kind of a menu, it is strongly dependent on the specific UI guidelines to look natural on the platform. Therefore neither code nor visual design is reusable in large portions.
- **Class Diagram Generator:** The class generation uses specific libraries to parse compiled java class-files. Therefore it's currently not portable to the web. If the code is changed to simple text parsing it could be written in a platform independent way.
- **File Drop:** This component enables the user to drag and drop uxf files into a diagram (the XML parsing is done by the I/O Handler). Like the listeners, the implementation is tied to the graphics library (Swing or the HTML5 File API 3.3).
- **Palette:** The palettes are predefined diagrams stored as uxf files and therefore are platform independent as long as they only contain platform independent GridElement.
- **Command:** These are simple commands using the Command pattern (Gamma et al. [39] categorize it as a behavioral pattern). Examples are Copy, Cut, Paste, AddElement, RemoveElement. The implementation of many commands depends on the platform like Copy and Paste, which is done using the Clipboard on the standalone version, but must use a different concept on the web version as described in 10.3. Other commands like AddElement can be implemented platform independently.

- **Selector:** The Selector holds the GridElement selection state for every diagram. There are several ways to select elements like clicking or lasso-selection (see the shortcut description in the UMLet wiki [46]).
- **Diagram:** The diagram is the hub of the application which connects most of the previously described components (e.g., a mouse click listener gets the GridElement which is located on the mouse-click location from the diagram).



Entry Point
- the point which starts the application
- e.g. Batch call
- e.g. Eclipse Plugin integration
- e.g. Swing start
reads
Configuration
- user specific settings
- Eclipse Plugin and Swing both use a properties file
Listener
- routes GUI Framework Events to Diagram
- e.g. MouseClickEvent
- e.g. KeyPressEvent
I/O Handler
- uses an XML parser
- transforms uxf into Diagram with GridElements
- transforms Diagram with GridElements into uxf
sends uxf to ▲
File Drop
- drag&drop uxf files into the drawpanel to create diagram
routes events
initializes
import/export into other format
interacts with ▲
Diagram
- contains diagram specifics (e.g. elements, diagram properties, zoom level ...)
- consists of DiagramHandler and DrawPanel
◂ interacts with
Menu
- offers basic operations
-e.g. undo, redo, open, save, ...
◂ creates
Class Diagram Generator
- generates class diagram from *.java or *.class files
executes on ▲
1
1
uses
1
is a
Command
- represents a undoable interaction with the diagram
- e.g. Add Element
- e.g. Move Element
Selector
- knows selection state of diagram
- can manipulate selection state
Palette
- predefined diagrams for certain use case
- e.g. UML Class, UML Package
0..1
selection state contains
contains
0..*
0..*
GridElement
- represents one element and its properties (text, color, ...)

Figure 4.1: High-Level Architectural Model of UMLet

Platform Dependency of the Model

Most components of the model work conceptually on different platforms, but have to be implemented in a platform specific way, because they depend on the platform API to work (e.g. listeners are bound to the UI API such as Swing or browser events).

However, partial reuse is often possible. E.g., the selection logic may be implemented in the shared codebase, but will still need a small platform specific part to redirect selection events from Swing or the browser to the shared code.

As mentioned before, the most important step is to make the GridElements as platform independent as possible, because this will result in the largest gain of shared code and it will improve the recognition value for UMLet on all platforms.

4.5 GridElement Architectural Model

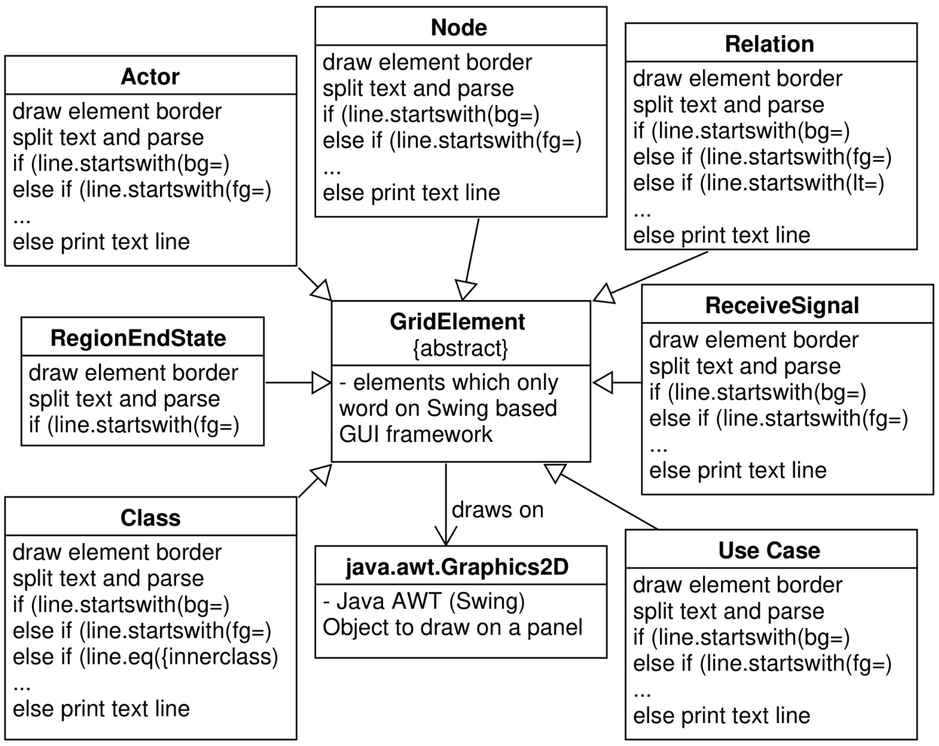


Figure 4.2: High-Level Architectural Model of the GridElements (not all elements shown)

As figure 4.2 shows, the GridElements don't have a clear structure. They share a parent class which contain some shared functionality, but the main functionality of parsing the text and translating functions is implemented as huge if-else-cascades which are copy&pasted between elements (e.g., relation has a paint method with 1000 lines of code). They are also coupled to the

Swing API and therefore to the platform, which means a major refactoring is required to make them reusable on other platforms, such as the web.

Goals of Redesigned GridElements Architecture

The redesign of the GridElements architecture is extensive and should achieve multiple goals.

Remove Duplicated Code in paint() Methods

Code duplication is a common habit, even in large software projects. The X Window System which had 714479 lines of code at the time of the analysis was analyzed using the program **Dup** (a program to find duplicated code). It found 2487 matches of at least 30 lines which involved 19% of the code [8].

The OldGridElements were created by many authors and some of them were external contributions. Therefore much of the element specific code is copy&paste code.

In the past users have reported bugs where certain features like **fg=** and **bg=** behaved differently depending on the element they were applied to. If these functions would be resolved in a common part of the code this couldn't happen.

While designing the NewGridElements, such redundancies should be avoided and a parsing algorithm must be created which is powerful enough to make these functions reusable between elements.

Separate GridElements from Swing API

The OldGridElement class extend javax.swing.JComponent and implement the paint(java.awt.Graphics) method to paint the content.

To unlink the GridElements from Swing without breaking the old elements, a GridElement interface should be introduced. The OldGridElement class still extend javax.swing.JComponent while implementing the GridElement interface. The NewGridElements will also implement the interface but not extend any GUI framework specific classes.

In addition an abstraction layer with generic draw methods should be introduced, to avoid direct coupling to Swing related objects and methods. This abstraction should be modeled as the DrawHandler.java interface, which must be implemented for every endpoint (e.g., Swing, GWT, ...). The interface should provide simple draw methods like drawLine(), drawRectangle() as well as draw-state-manipulations like setting colors, line thickness and more.

Separate Properties Parsing and Drawing Calls

Another problem of the OldGridElement is that the functions in the properties of an element are parsed every time the GUI framework makes a paint-call on the element.

This is especially problematic for elements with complex property functions and a time consuming parsing process (e.g., the PlotGrid, the All-in-one diagrams or word-wrap calculations)

The solution to the problem of time consuming functions is a separation of the paint-method into:

1. an **update()** method which parses the properties, analyzes the functions and calculates a derived state. It is called every time the properties change.

2. a **paint()** method which maps the derived state into draw-calls to the generic DrawHandler. It does not do complex calculations based on properties on every draw call.

Keep Backwards Compatibility

An important goal is to preserve backwards compatibility to the OldGridElements and the current platforms. Therefore from a users perspective, the mentioned changes must be implemented as additional features without removing existing functionality.

However it is possible to implement small changes if its beneficial to the user experience. For example, the current selection logic is complicated and the selection state is shared between GridElement and Selector classes. It is OK if this logic is simplified and generalized, if it results in a more intuitive and consistent user experience.

Auto-completion for Function Names

Currently the only way to discover a previously unknown function for a GridElement is to find an element which uses it. Typically the user needs a certain feature (e.g., mark a UML class as active) and searches the palette for it. If he finds an element which looks like its using the feature, the user checks its properties and has to guess which line is the function to trigger this feature.

Although this is typically a simple and intuitive process, the usability can be further improved by offering autocomplete for all known functions. This is an alternative way of discovering the available functions which is especially helpful if the user cannot remember the exact syntax but already knows that such a feature exists.

Autocomplete should be triggered with a certain keyboard shortcut which lists all available functions for the current element. It should also show a description of the effect of the function.

Improved GridElement Architectural Model

The first step of improving the model is to identify reusable parts of the old model which should be shared. Figure 4.3 colorizes the architectural model from figure 4.2 to show those parts:

- Every element implements a parser (green color)
- Some functions work on most elements (blue, red)
- Other functions work only on few or one element (yellow, pink)
- Most elements print properties which are not functions as text (cyan)

Those shared parts can be encapsulated into a wrapper object which links a specific command (e.g. fg=red) to a specific action (e.g. change the foreground color to red). These wrapper objects are called Facets. Every element lists applicable facets and during the parsing procedure

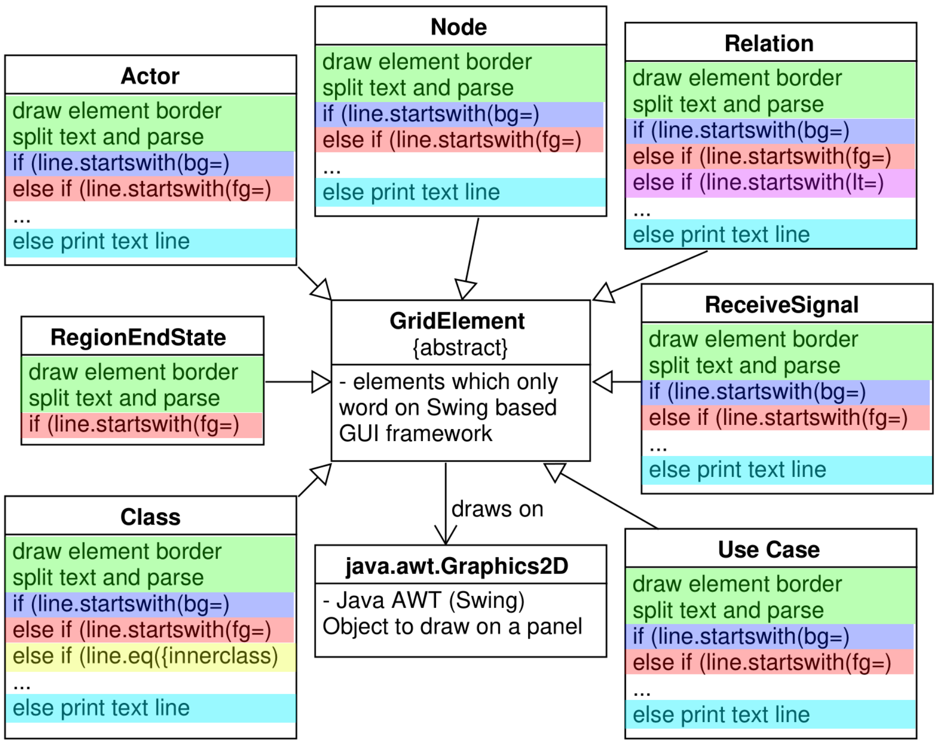


Figure 4.3: Shared parts of GridElements model

of an elements properties text, the parser checks if any facet triggers and calls its handleLine method.

Figure 4.4 shows how those shared facets can be modelled. For example both Class and Use Case can change the background color, therefore both use the BgColorFacet, but only the Class can have an inner class, therefore it's the only element which uses the InnerClassFacet. During the parsing, the GridElement passes the usable facets to the parser which checks for every facet if it triggers and calls its handleLine method.

This modularization is the first important step to make elements work on all platforms and reuse as much code as possible. The next step is to introduce a custom interface for drawing, which must be implemented by each platform to guarantee platform independence of the shared codebase. Finally the GridElement requires an additional abstraction to let OldGridElements (the legacy elements) and NewGridElements (elements designed with the new architecture in mind) coexist. This is important, because it's not feasible to change the old elements, because there are many minor differences, intentionally or unintentionally, which should be kept for old elements (to avoid breaking old diagrams) but changed for new elements (to simplify the code and streamline the expectations of users).

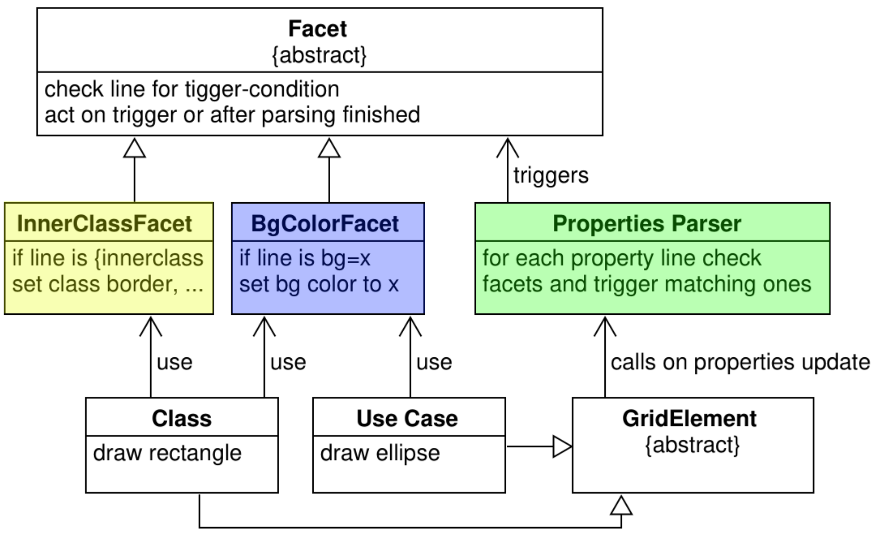


Figure 4.4: Shared parts of GridElements expressed with facets (excerpt)

Description of the New Architectural Model

The comprehensive new architectural model is shown in figure 4.5 and can be categorized into 3 groups.

Model elements to analyze properties and handle text based functions

- **Properties Parser:** Anytime the properties of an element change, the parser reads the new properties and parses them according to the Facets of the **GridElement**. Additional info can be found in chapter 5.2.

- **Facet:** Technically a Facet is an interface which specifies a method which checks if the facet triggers (typically by checking if a specific function-text is part of the current line). Furthermore it consists of a method, which will be executed if the facet triggers, an auto-completion text which describes the functions name, parameter and action, and another method which is called when parsing is finished, because some actions should only happen at the end of the parsing procedure.

  For example, the ForegroundColorFacet will trigger on the text **fg=<color>** and will transform the given <color> into a known color-object which will then be set as foreground color in the **DrawHandler**.

Model elements to abstract old and new GridElements (for backwards compatibility)

- **GridElement:** This class is now an interface to abstract the platform-independent newly designed NewGridElements from the OldGridElements which are only maintained for backwards compatibility.

- **OldGridElement:** To avoid breaking backwards compatibility for old diagrams, all of the old elements will remain (mostly) unchanged and therefore will still work as expected on the Swing platform. After some UMLet releases, the OldGridElements will be removed. Section 10.2 discusses several approaches for the removal.

- **Custom and All in One Elements:** These elements are currently not available in a platform independent version and should be migrated in the future (see chapter 10.1).

- **NewGridElement:** This is the new parent class for any platform independent GridElement. Each subclass specifies some immutable element characteristics and a set of Facets. For example, an UML Use Case has an ellipse as border and applies all basic Facets (fg-color, bg-color, …). In addition the class contains some basic methods like setting and getting properties, getting the element size and more.

- **Stickable and Relation:** The old Relation class was distributed all over the other components (using **instanceof** checks) to incorporate the special handling of relations. The new concept uses a minimal interface called Stickable. Any class which implements this interface will be able to stick to other elements if they are moved (currently only the new Relation implements this interface).

Model elements to abstract underlying graphical framework from the GridElements

- **DrawHandler:** This is an abstract class and the only way for an element to draw something on the screen. It offers basic drawing-operations like drawLine(), drawEllipse(), printText(), setForegroundColor() and so on. The class must be subclassed for every platform to implement those basic drawing methods.

- **DrawHandlerSwing and DrawHandlerGwt:** These are the concrete subclasses for each of the target graphics library (Swing or Html5 Canvas).

- **Component:** The Component links the platform-agnostic GridElement to a specific platform. The Component contains the platform-specific part and is injected into the GridElement as a Constructor argument. The most important content of the Component is the concrete implementation of the DrawHandler which is made available to the GridElement with a getDrawHandler() method.

- **ComponentSwing and ComponentGwt:** These are the concrete subclasses for each of the target graphics library (Swing or Html5 Canvas).

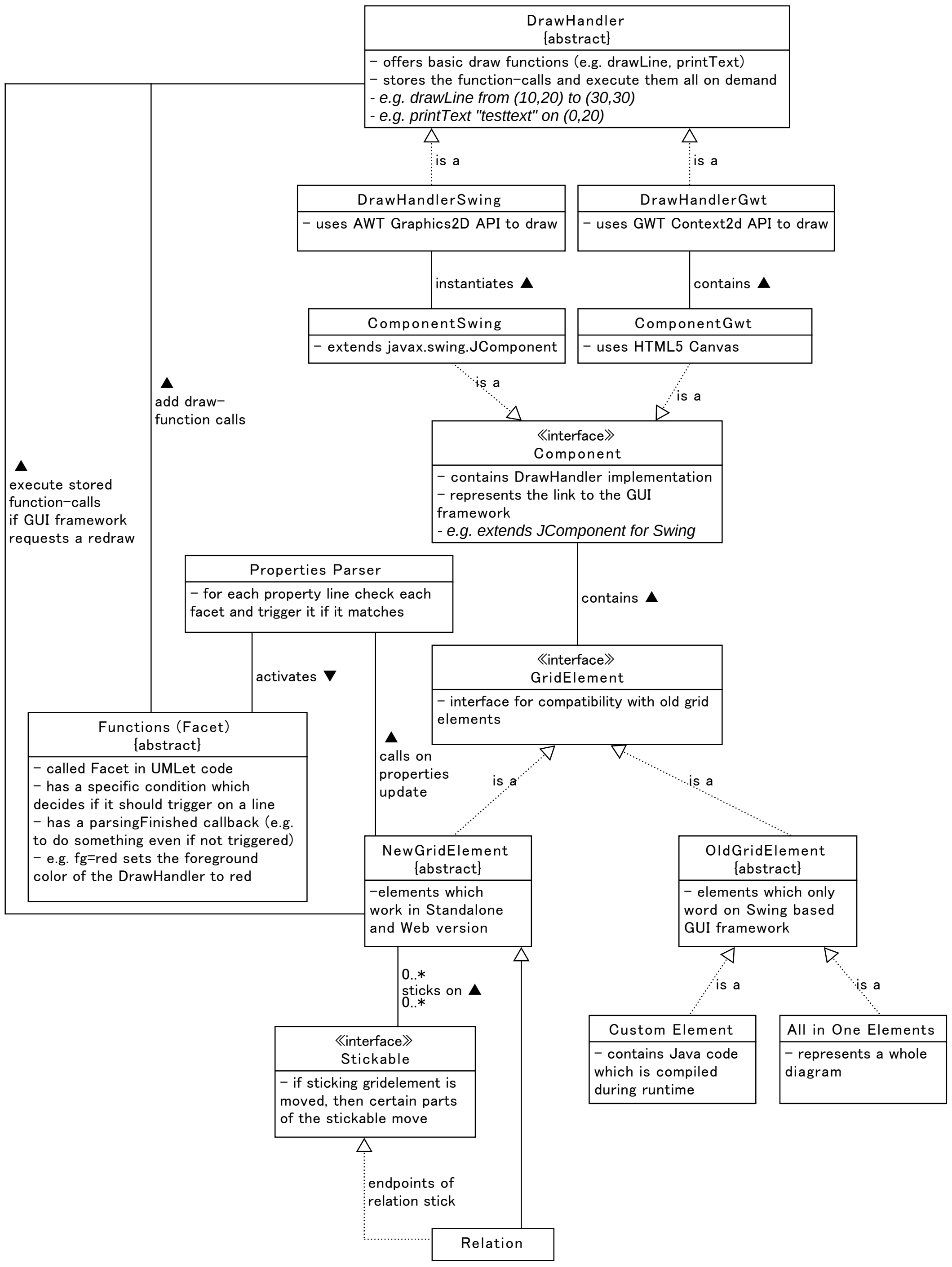

DrawHandler
{abstract}
- offers basic draw functions (e.g. drawLine, printText)
- stores the function-calls and execute them all on demand
- e.g. drawLine from (10,20) to (30,30)
- e.g. printText "testtext" on (0,20)
is a
is a
DrawHandlerSwing
- uses AWT Graphics2D API to draw
DrawHandlerGwt
- uses GWT Context2d API to draw
instantiates ▲
contains ▲
ComponentSwing
- extends javax.swing.JComponent
ComponentGwt
- uses HTML5 Canvas
is a
is a
▲ add draw-function calls
▲ execute stored function-calls if GUI framework requests a redraw
«interface»
Component
- contains DrawHandler implementation
- represents the link to the GUI framework
- e.g. extends JComponent for Swing
Properties Parser
- for each property line check each facet and trigger it if it matches
contains ▲
activates ▼
«interface»
GridElement
- interface for compatibility with old grid elements
Functions (Facet)
{abstract}
- called Facet in UMLet code
- has a specific condition which decides if it should trigger on a line
- has a parsingFinished callback (e.g. to do something even if not triggered)
- e.g. fg=red sets the foreground color of the DrawHandler to red
▲ calls on properties update
is a
is a
NewGridElement
{abstract}
-elements which work in Standalone and Web version
OldGridElement
{abstract}
- elements which only word on Swing based GUI framework
0..*
sticks on ▲
0..*
is a
is a
«interface»
Stickable
- if sticking gridelement is moved, then certain parts of the stickable move
Custom Element
- contains Java code which is compiled during runtime
All in One Elements
- represents a whole diagram
endpoints of relation stick
Relation

Figure 4.5: Detailed model of the redesigned GridElements and their environment

4.6 UMLetino (Web) High-Level Architectural Model

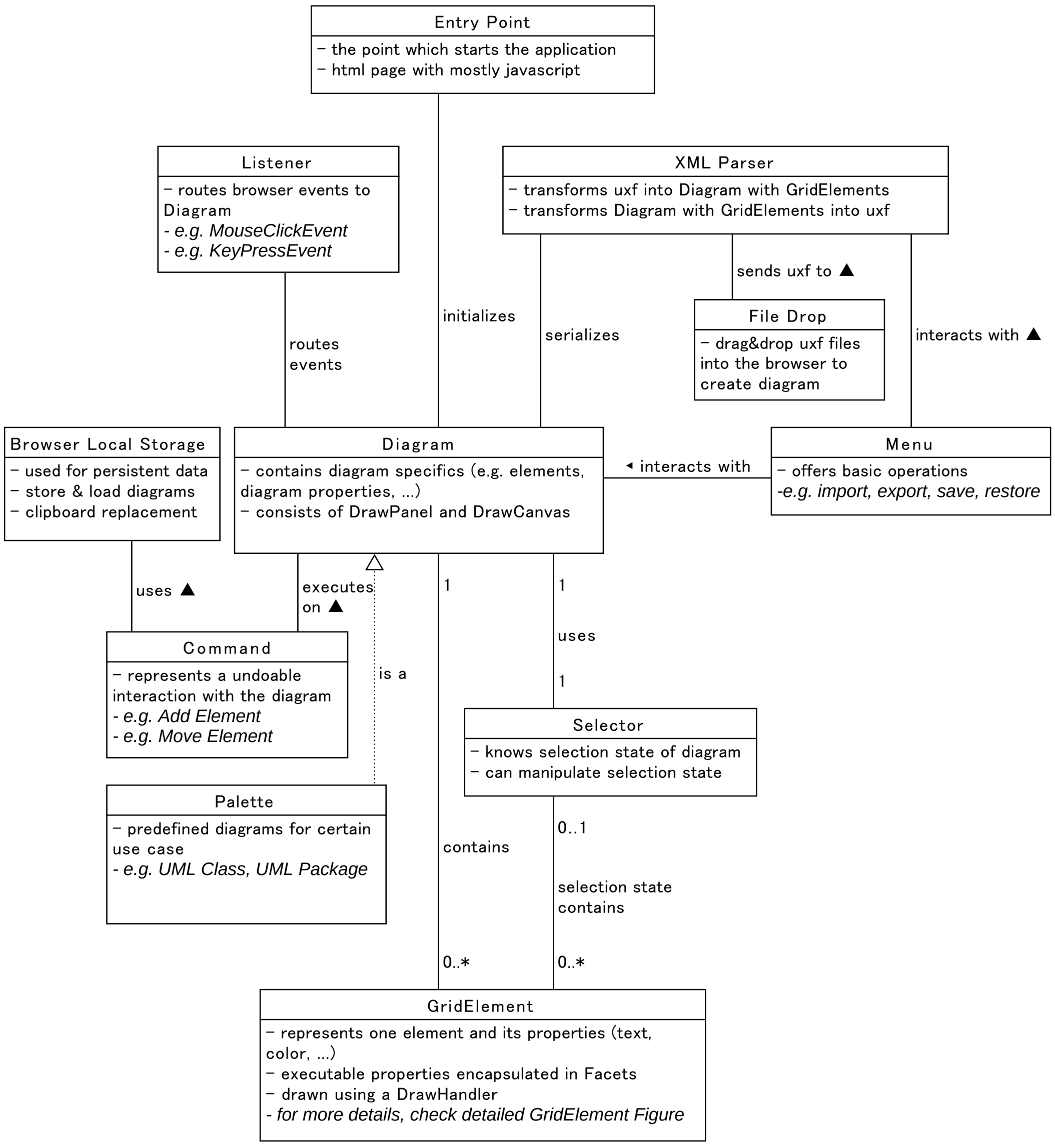
In addition to the improved platform-agnostic GridElement Model, a new web-specific model has been created which is used to implement the web version (see figure 4.6). Fortunately most of the current model can be reused, but some changes were necessary due to platform differences.

The following elements have been removed from the model

- **Configuration:** is currently not planned to keep the first version of UMLetino simple.
- **Class Diagram Generator**: Cannot be ported, because it uses libraries which would not work with GWT (such as a library to analyze the Java byte-code).

The following elements have been added to the model

- **Browser Local Storage:** is an important part of the web version, because it opens a way for a quick save and load mechanism and for some kind of an alternative clipboard.
- **XML Parser:** the I/O Handler has been changed to a simple XML Parser, because there is only a very limited file interaction possible. Unfortunately the XML Parser cannot be the same as in the standalone version, because the GWT XML API uses other classes than the typical Java XML API. Another issue is that the XML format of the web parser is simpler because it must not account for the slightly different syntax of OldGridElements.



Entry Point
– the point which starts the application
– html page with mostly javascript
Listener
– routes browser events to Diagram
- e.g. MouseClickEvent
- e.g. KeyPressEvent
XML Parser
– transforms uxf into Diagram with GridElements
– transforms Diagram with GridElements into uxf
sends uxf to ▲
File Drop
– drag&drop uxf files into the browser to create diagram
initializes
routes events
serializes
interacts with ▲
Browser Local Storage
– used for persistent data
– store & load diagrams
– clipboard replacement
Diagram
– contains diagram specifics (e.g. elements, diagram properties, ...)
– consists of DrawPanel and DrawCanvas
◂ interacts with
Menu
– offers basic operations
-e.g. import, export, save, restore
uses ▲
executes on ▲
is a
1
1
uses
1
Command
– represents a undoable interaction with the diagram
- e.g. Add Element
- e.g. Move Element
Selector
– knows selection state of diagram
– can manipulate selection state
Palette
– predefined diagrams for certain use case
- e.g. UML Class, UML Package
contains
0..1
selection state contains
0..*
0..*
GridElement
– represents one element and its properties (text, color, ...)
– executable properties encapsulated in Facets
– drawn using a DrawHandler
- for more details, check detailed GridElement Figure

Figure 4.6: High-Level Architectural Model of UMLetino

CHAPTER 5

Project Modularization

This section describes the steps to achieve **(M1) Project Modularization** from section 1.3, which is necessary to prepare the codebase for new platforms. The goal is to create a platform independent program core, which can be reused by several platform specific endpoints.

The modularization is structured into several sub-tasks, such as the introduction of reusable functions, which get consumed by a generic parser, an independent drawing API to abstract the underlying graphics framework, and the creation of separate Eclipse projects.

Finally further improvements due to encapsulation of element behavior are presented and the results of the modularization are discussed.

5.1 Functions (Facets) as Central Concept

As described, UMLet uses predefined functions (see chapter 3.6), which can be entered into the properties text of an element, to change the appearance or behavior of the element.

The program analysis has shown, that there are many general purpose functions. The user expects that a function works on all elements where it makes sense (e.g., an element without lines will not offer line-customization functions)

To improve the usability of UMLet, it is important that a specific action always uses the same text to trigger a function on all elements. This was not always the case for the old elements which is shown by the following example:

Different Ways of Setting Text Alignment in OldGridElement

- **Component:** its text is horizontally and vertically centered by default, but if the properties text starts with an inverted comma, the whole text is moved to the top left corner.

- **Package:** the text is located at the top left corner and individual lines can be centered horizontally by using the prefix **center:**.

To avoid those differences in the new implementation, table 5.1 describes several general purpose functions, which will work the same for all elements.

Function Name(+Param)	Description (Action)
bg=<color>	set the background color
fg=<color>	set the foreground color
style=autoresize	the element automatically resizes if text exceeds the border
style=wordwrap	linebreaks are inserted if text exceeds the border
style=noresize	the text is not visible if it exceeds the border
fontsize=<float>	set the font size
group=<integer>	set the group (groups will always be selected together)
halign=left | center | right	set the horizontal alignment of printed properties text
layer=<integer>	set the layer (visibility and selection priority for overlapping)
lt=. | .. | -	set the line type (dashed, dotted or solid)
lw=<float>	set the line width
--	draw a horizontal separator line on the current text position
valign=top | center | bottom	set the vertical alignment of printed properties text

Table 5.1: General purpose functions (name, parameter and description of the action)

Facets encapsulate functions text, action and documentation

Although many of the functions in table 5.1 already work with the old elements, their implementation is completely separate. This results in bugs where a seemingly shared functionality behaves differently, depending on the element it is applied to. To fix this problem, the new architecture uses the concept of Facets, which are fine granular classes, which trigger on a specific text and execute some code. In other words they encapsulate the text to trigger, the action (code which should be executed when it triggers) and the documentation (for auto-completion) of a function.

Every facet has to implement the abstract class **Facet**, which contains the following code:

```
/**
 * A Facet is a simple handler method which acts on certain lines
 * and does a specific job if it should act.
 * It is important that Facets are ALWAYS STATELESS.
 * If any State is required, it should be stored using the
 *  {@link PropertiesParserState#getOrInitFacetResponse(Class, Object)} method
 */
public abstract class Facet {
  /**
   * @param line the current line which is parsed
   * @param state the current state of the parser
   * @return true if the handleLine() method of this facet should be applied
   */
  public abstract boolean checkStart(String line, PropertiesParserState state);

  /**
   * This method is invoked at the time when a specific line is parsed
   * @param line the current line which is parsed
   * @param state the current state of the parser
   */
  public abstract void handleLine(String line, PropertiesParserState state);

  /**
   * @return a list of objects where each one represents one line for autocompletion
   */
  public abstract List<AutocompletionText> getAutocompletionStrings();

  /**
   * This method is called once for every Facet AFTER all lines of text has been parsed
   * E.g. useful for facets which collect information with every line but need complete
   * knowledge before they can do something with it
   *
   * @param state the current state of the parser
   * @param handledLines the list of lines this facet has been applied to
   *        (in the order of the handleLine calls)
   */
  public void parsingFinished(PropertiesParserState state, List<String> handledLines) {
    // default is no action
  }

  /**
   * facets with higher priority will be applied before facets with lower priority:
   * The order is: For all lines
   * 1. Check all First-Run Facets from HIGHEST ... LOWEST
   * 2. Check all Second-Run Facets from HIGHEST ... LOWEST
   */
  public Priority getPriority() {
    return Priority.DEFAULT;
  }

  /**
   * The parser runs twice. Facets where this method returns true, are part of the first run,
   * other facets are part of the second run
   *
   * Facets of the first run will influence the whole diagram, even if they are located at the bottom.
   * e.g. bg=red must be known before drawing the common content of an element; style=autoresize
   * must be known as soon as possible to make the size-calculations
   *
   * Facets of the second run have less side effects (e.g. printText just prints the current line,
   * -- transforms to a horizontal line at the current print-position)
   */
  public boolean handleOnFirstRun() {
    return false;
  }
}
```

One simple example is the **SeparatorLineFacet**, which transforms **--** to the action **draw a horizontal line**. The 2 relevant method implementations are:

```
private static final double Y_SPACE = 5;

public boolean checkStart(String line, PropertiesParserState state) {
    return line.equals(--);
}

public void handleLine(String line, PropertiesParserState state) {
    DrawHandler drawer = state.getDrawer();
    double linePos = state.getyPos() - drawer.textHeight() + Y_SPACE / 2;
    XValues xPos = state.getXLimits(linePos);
    drawer.drawLine(xPos.getLeft() + 0.5, linePos, xPos.getRight() - 1, linePos);
    state.addToYPos(Y_SPACE);
}
```

The **checkStart(...)** method simply means that every line which equals the key will trigger this transformation.

handleLine(...) is a bit more complex as there are different element shapes (e.g., a **Class** is a rectangle, a **Use Case** is an ellipse), but using the available data from the parser state the necessary coordinates for the line can be calculated in a generic way by using the **getXLimits** method.

5.2 New Properties Parser

The core of UMLet is the mapping of a textual to a graphical representation. The previously defined shared functions need a way to connect with the properties parsing process.

Therefore a generic parser must be implemented. Figure 5.1 described the whole parsing procedure. It consists of a pre-parsing to do calculations and a real parsing phase which draws the element. Each of those phases consists of a first and a second parser run.

Figure 5.2 shows what happens during each parser run.

For every line of the properties, all first-run-facets of the current element are checked with the method **facet.checkStart(String line, PropertiesParserState state)**. As soon as one facet returns true, its **facet.handleLine(String line, PropertiesParserState state)** method is called and no other facet can trigger on this line.

After all properties lines are checked, the **facet.parsingFinished(PropertiesParserState state, List<String> handledLines)** method is called for all facets. As a parameter, the lines to which this facet has been applied to, is given. Therefore some facets can execute a default behavior if they are not triggered at all, and other facets can do some things which are not possible on a line-by-line basis (e.g., there is a facet which collects all lines before the first separator line **--** and puts them into the top left corner of the package-element and the rest into the package-element content body)

Detailed Description

Pre-Parser Phase

The first step of the parser is the pre-parsing phase, which works exactly as the real parser phase (first run, drawing common content and second run), but without issuing any real draw-functions. It uses a DrawHandler (see section 5.3) implementation, which does not draw anywhere (its drawing methods do nothing), but otherwise behaves like a real DrawHandler (e.g., it can calculate the width of a text).

This phase is currently only used to calculate the expected size of the properties text which is printed as text into the element (i.e. how many pixels does the text block require to be displayed correctly).

Some facets manipulate the dimensions when they run. Some examples are:

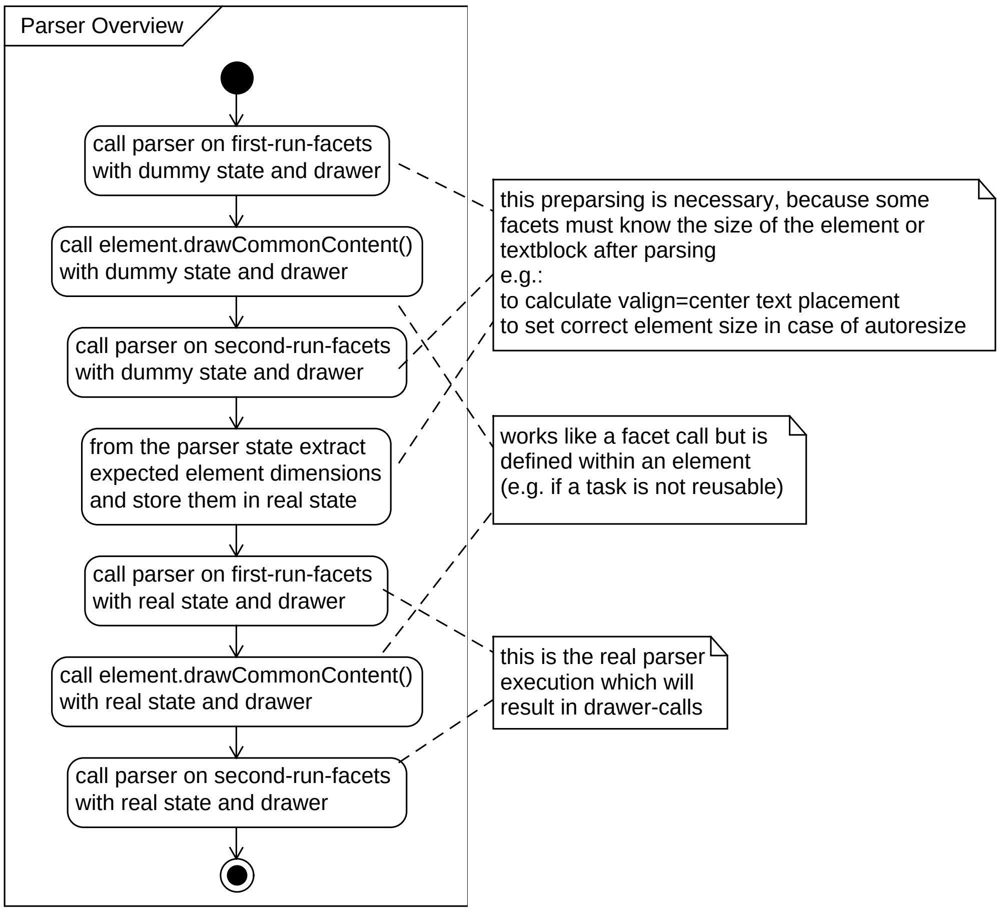

Parser Overview
call parser on first-run-facets with dummy state and drawer
call element.drawCommonContent() with dummy state and drawer
call parser on second-run-facets with dummy state and drawer
from the parser state extract expected element dimensions and store them in real state
call parser on first-run-facets with real state and drawer
call element.drawCommonContent() with real state and drawer
call parser on second-run-facets with real state and drawer
this preparsing is necessary, because some facets must know the size of the element or textblock after parsing
e.g.:
to calculate valign=center text placement
to set correct element size in case of autoresize
works like a facet call but is defined within an element (e.g. if a task is not reusable)
this is the real parser execution which will result in drawer-calls

Figure 5.1: Overview of the properties parser

- the SeparatorLineFacet adds a few pixels of additional space before the next text line is printed.
- the TextPrintFacet prints a line and adds the text height, plus an additional vertical space, to the text block dimensions.

These examples show, that it is required, that all facets have been checked before the final text block dimensions are known.

The problem is that the text block dimension (after applying all facets) is already needed by some facets.

- If **valign=center** or **valign=bottom** (see table 5.1) is used, the TextPrintFacet needs to know the text block height in order to calculate the required distance between the upper border of the element and the first text line to draw.

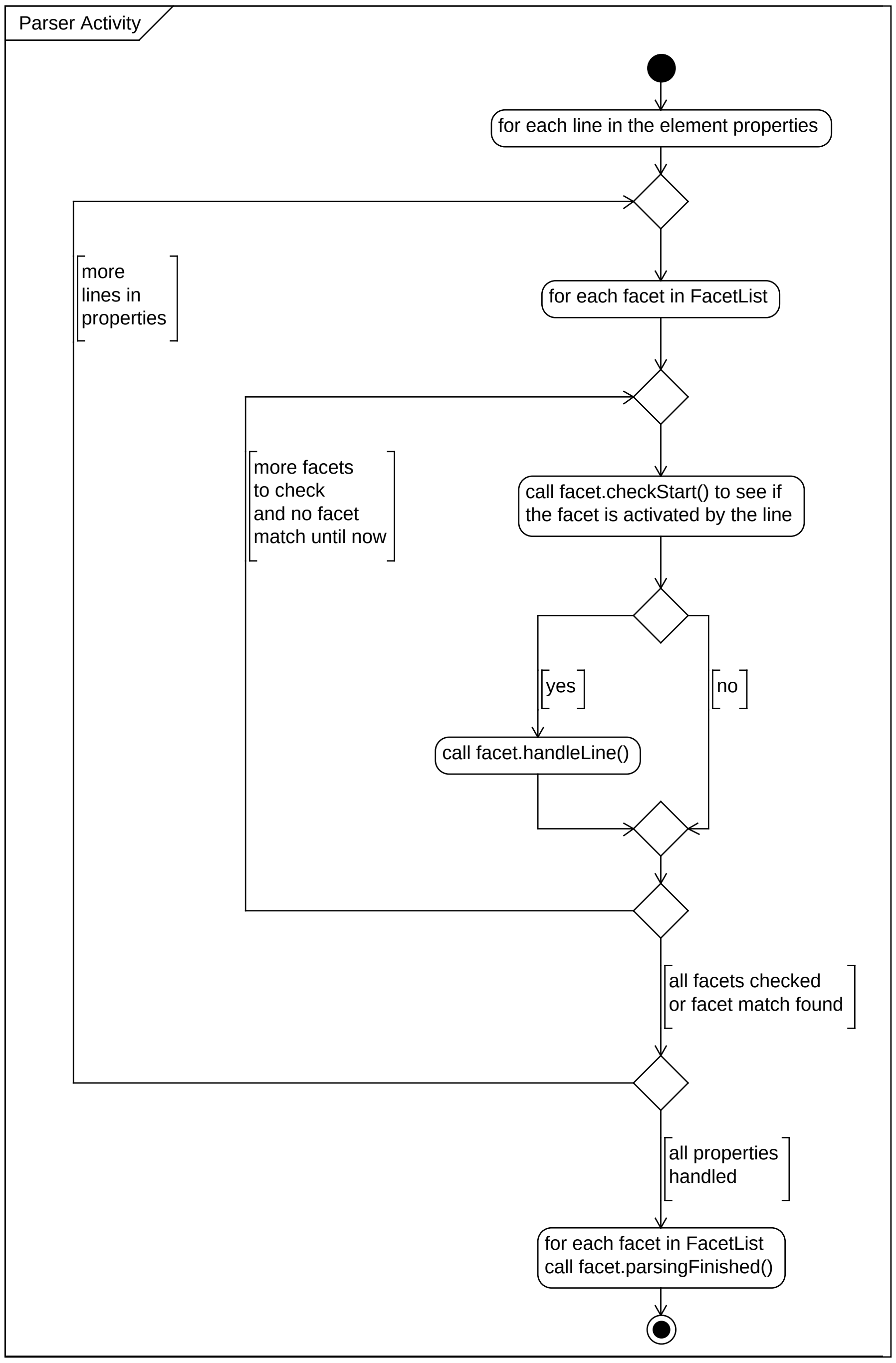

Parser Activity
for each line in the element properties
more lines in properties
for each facet in FacetList
more facets to check and no facet match until now
call facet.checkStart() to see if the facet is activated by the line
yes
no
call facet.handleLine()
all facets checked or facet match found
all properties handled
for each facet in FacetList call facet.parsingFinished()

Figure 5.2: Details of one parser run

- If **style=autoresize** is used, the element automatically resizes itself to the minimal required size to fit the whole properties text block into the element. This resizing must be done before applying any facet which would depend on the correct element size (e.g., if a facet draws a rectangle around the element, the size of the element must be fixed - which means autoresizing must be finished - before the facet is parsed)

These examples show that every facet can potentially manipulate the text block size, and every facet can depend on the actual text block size after parsing, therefore a full parsing of all properties with all facets must be done without any drawing (just for text block size calculation), before the real parsing with drawing can be done.

Real Parser Phase

After the pre-parsing phase, the parser state is reset, the expected text block dimensions are set into the parser state and the real parser phase is executed, using a real DrawHandler (which draws on a Swing panel, a HTML canvas, . . .).

There are intentionally no other differences to the pre-parser phase, to make sure the expected text block dimensions stay the same.

Parser First Run

Each of those two parser phases (pre-parser and real parser phase) parses the properties text two times (first parser run and second parser run). Every facet belongs to either the first or second run, but not both.

The reason for this separation is that some facets must be applied before other facets, even if the text which triggers the facet appears in a later line of the properties.

For example the following properties text translate to figure 5.3.

```
SimpleClass
--
text
fg=red
style=autoresize
```

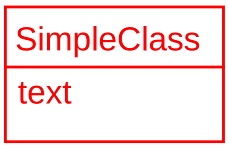


Figure 5.3: Parser GridElement UML Class example

The facets which are registered for the UML Class element execute the following actions:

- **SimpleClass** is drawn into the element
- **--** draws a horizontal line
- **text** is drawn into the element
- **fg=red** changes the foreground color to red
- **style=autoresize** makes sure that the element changes size to fit the text dimensions

If the text would be parsed sequentially, the text and the horizontal line would not appear as red, because **fg=red** appears after the other lines. Also it is unclear if the element border should be colored red.

To fix those problems, the properties text is parsed two times (as shown in figure 5.2) and every facet must register for the first or second run.

1. run parser with first-run-facets (such as foreground color or autoresize)
2. draw common content (such as the element border)
3. run parser with second-run-facets (such as drawing text and horizontal lines)

The foreground color is changed before anything is drawn, therefore the text, horizontal line and border are all colored red as expected. The autoresizing is also applied before the horizontal line or border are drawn, therefore the dimensions are calculated correctly.

Draw GridElement Common Content

After handling the first-run facets, the **element.drawCommonContent(PropertiesParserState state)** method on the GridElement is called which can draw some element specific content, e.g., the rectangle around the element.

This step of the parser is optional and only used by some elements, but it is a way to draw content which is only relevant for specific GridElements without the occurrence of a specific line as a trigger (e.g., every Use Case has an ellipse as border, even if the properties text is empty)

The equivalent to this is a First-Run-Facet which acts on the **parsingFinished(...)** method, because then it will be executed, even if the properties are empty.

Parser Second Run

The second run is equivalent to the first run, but checks only the second-run-facets. The most important second-run-facet is the TextPrintFacet which simply prints any property line as a text within the element.

5.3 Platform Independent Drawing API (DrawHandler)

One goal of UMLet's redesign is to decouple the GridElements from the platform. The major challenges of this goal are:

- Elements should look the same on all platforms.
- Diagram files should work on all platforms.
- A uniform API should hide platform specific quirks.

Some examples for **platform specific quirks** are:

- Html canvas must start at half pixel for crisp lines (otherwise they look blurry).
- Swing cannot draw on the rightmost/bottom pixel of an element.
- PDF exports cut lines which start at 0. pixel.

The goal of the new drawing API is to hide those from the developer, i.e., the html canvas must handle the half-pixel displacement within its implementation, the PDF implementation must move lines which are drawn on the 0. pixel to the 1. pixel, and so on.

Current Codebase

The OldGridElement class extends **javax.swing.JComponent** and is therefore coupled to the Swing GUI framework. This makes it impossible to reuse the elements on platforms which do not support this framework (such as GWT and Android).

In case of GWT the emulated JRE classes [29] do not contain any Swing or AWT classes. Although it is possible to implement missing classes (as described in the GWT developer guide [94] in section **Overriding one package implementation with another**) this is limited to GWT, because the code is compiled to JavaScript. Other platforms without Swing support like Android wouldn't work.

In addition to the missing implementation, there are other problems with the OldGridElements (such as the differences in handling functions as described in section 5.1), therefore the parsing and painting process must be redesigned.

Fortunately the work on Plotlet as part of a bachelor thesis [37] already introduced the concept of an abstract Drawing API. This concept will now be expanded to every drawing call of the NewGridElements (the OldGridElements which remain unchanged for backwards compatibility and removed sometime in the future).

This Drawing API is an abstract class called DrawHandler which must be implemented by every target platform.

The Main Purposes of the DrawHandler are:

- storing the current draw style like foreground color, the background color, line width, line thickness, font size, font family, ...
- offering basic drawing-operations like drawLine(), drawRectangle(), drawEllipse(), printText(), ...
- offering calculation methods like textWidth(), textHeight(), getDistanceBorderToText(), getDistanceBetweenTextLines(), ...
- zooming the coordinates if necessary (the OldGridElements used to apply the zoom factor manually for every single drawing call which is error prone and bad design as zooming should happen transparently on a higher layer).
- acting as an abstraction layer to implement OS or browser specific behavior (e.g., although browsers use different JavaScript functions to draw dashed lines, this inconsistency must only be handled within the DrawHandler and not in every element which draws dashed lines).

When a drawing-operation (drawLine, . . .) is called, it does not immediately issue method calls to the underlying GUI framework. Instead it copies the current style (foreground color, background color, line thickness, . . .) and instantiates a Java Runnable, which is implemented to call the required methods of the GUI framework based on the copy of the current style.

In other words it emulates a function with its closure (the style at the time of the method call), which can be called at any time in the future to reproduce the same result (e.g. draw a red line even if the foreground color has changed to blue at a later point in time). The DrawHandler offers a method to draw all those stored functions.

Therefore the DrawHandler stores all necessary method calls to the underlying GUI framework in order to redraw the element at any time. Simply put, it contains the visual representation (the draw-model) of the elements textual properties. Recalculation is only necessary when the state of the element (properties or size) changes.

The main advantage of this approach besides the platform independence is the clear separation of parsing and drawing into two steps:

1. Parse the properties and create a draw-model.
2. Draw the stored model on a canvas.

This separation avoids problems which some of the old elements have [1]. The issue describes that the all in one diagrams calculate their real size after drawing, therefore it seems to work if multiple calculations happen within a short time frame (e.g., while changing the properties) but it doesn't work if there is only one execution of the paint() method.

[1]issue about all in one diagram in UMLet: `https://github.com/umlet/umlet/issues/159`

5.4 Platform Independent Basic Classes

Some classes which are used for drawing are specific for a certain GUI Framework. These classes must be replaced by own implementations which work on any target platform.

Color

One example is the Color class which exist in several variations:

- **AWT/Swing:** java.awt.Color
- **SWT:** org.eclipse.swt.graphics.Color
- **GWT:** com.google.gwt.canvas.dom.client.CssColor

As the DrawHandler is a reusable part which must know about colors, a single Color class is created (called ColorOwn to make it explicit even without import statements). Every time a new platform is implemented, a converter from ColorOwn to the platform specific color-class must be implemented too.

The advantage of this approach is that the conversion is transparent for the developer of GridElements, because every drawing call must use the DrawHandler which only knows about the ColorOwn class.

The separate ColorOwn class also has the advantage of providing several predefined colors as class constants. Also the API of the class is the same for every platform.

Geometric classes

As a drawing program, UMLet needs several mathematical functions like distance between lines, calculating the slope of a line and more.

Unfortunately most of the classes for this purpose are also GUI framework specific (e.g., **java.awt.Point** or **java.awt.geom.Line2D**) and do not exist in GWT.

Therefore the same approach as before is used and own classes for Points, Lines, Rectangles, . . . are created.

The main reason for this is to encourage object oriented programming (e.g., a Line consists of a start-point and an end point instead of four fields x1, x2, y1, y2). These classes also offer several mathematical functions like line.getAngleOfSlope(), line.getDistanceToPoint(point), point.distance(point), . . .

As these classes can be changed by the developer, there is always an obvious location to search for and implement such mathematical functions which avoid redundant code which could happen if these functions would be located in different utility classes.

Another advantage is that the precision of drawings can be improved, because the line class uses points with **double** precision instead of **int**. Therefore no calculation before the final draw-call is rounding numbers.

5.5 Separate Eclipse Project for Shared Code

The previous steps improved the re-usability of the most important components of UMLet.

The next step is to extract the reusable code into a separate a separate Eclipse Project, which is used by both endpoints.

Another advantage of this code separation is the introduction of simple code access rules. Code from the shared project should never be able to use any platform specific code, and this behavior is enforced by the dependency management of Eclipse as soon as they are separate projects with a one-way dependency (platform specific project depends on the shared project but not the other way around).

At a later point in time, the projects will be migrated to the build and dependency management tool Apache Maven (see section 8.1), which improves and automates the build process and standardizes the project structure.

5.6 Encapsulation of Element Behavior

Gamma et al. [39] define the term **encapsulation** as *"The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object."*

It is an important design principle of object-oriented programming which was violated in UMLet by OldGridElements. Implementation details leaked out of the classes on several locations.

UMLet often uses instanceof checks to implement element specific behavior. E.g. listener classes check the type of the element and execute element specific code:

```
if (element instanceof Group) { group_specific_code }
if (element instanceof Relation) { relation_specific_code }
```

To avoid these **instanceof** checks, the element specific code must be moved into the implementation class of the element itself.

Some concrete examples of those checks are described in the following subsections.

Group

Groups were designed as specific GridElements which can contain other GridElements. This design has led to several issues and it was unnecessarily complex.

Therefore the concept of groups has been redesigned to a simple condition: *if one element of a group is selected, select all other elements of the same group automatically.*

That means it's no longer necessary to handle them as a specific type of GridElement, instead a grouped element stays the same and the only external part which knows about grouping is the **Selector**, which is the class which handles selections. The **Selector** simply checks if the selected element is grouped, and if it is, it selects all other elements within the same group.

Therefore the behavior of groups have been encapsulated, as only the GridElements know about their groups and the only external instance which uses the grouping information is the **Selector** which resolves the expected behavior.

Relation

Click listeners act differently to add, remove or move relation points. The easiest way to solve the issue is to move the action which will be executed on a click into the GridElement class itself. Therefore GridElements simply expose a **drag()** method which is called if a drag-action was executed by the user, but the element itself specifies how to handle such a drag command (e.g., most elements simply move according with the mouse cursor, but a relation element can check if the mouse was on a relation line and move the line accordingly)

Selection prioritizes Relations over other elements. This problem can be solved by introducing a new generic concept of layering elements. Many users have asked for a way to specify the layering (z-order) or elements and it can be implemented in a generic way. Every GridElement can specify its layer (using the layer=<integer> function which is listed in table 5.1). The layer not only specifies which elements have a higher selection priority in case of overlapping elements, but it also specifies which element overlaps another element in case of background colored drawings.

Code to handle sticking of relation ends is scattered through the codebase. This can be resolved by introducing a **Stickable** interface, which can be implemented by any element which should stick to another element during move-commands.

The code to handle sticking is moved to the GridElement base class, therefore the platform specific code is not required to know any details about it.

The only code which could not encapsulated within the GridElement or Relation class is that the diagram must know which elements should stick and when to disable or enable sticking, but the concept of sticking has been leveraged and is not only reserved for Relations anymore.

5.7 Results

While the OldGridElements were each tied to a specific use case and created mostly by copying code from one element to another, the NewGridElements are designed as multiple purpose elements, which can fit several roles as long as the user configures them appropriately. E.g., the special states of an UML Activity diagram (final state, initial state, ...) are one type which is specified using **type=initial**, **type=final**, ...

Also code which is specific for certain GridElements have been encapsulated in the GridElement class instead of spreading it over the whole source code.

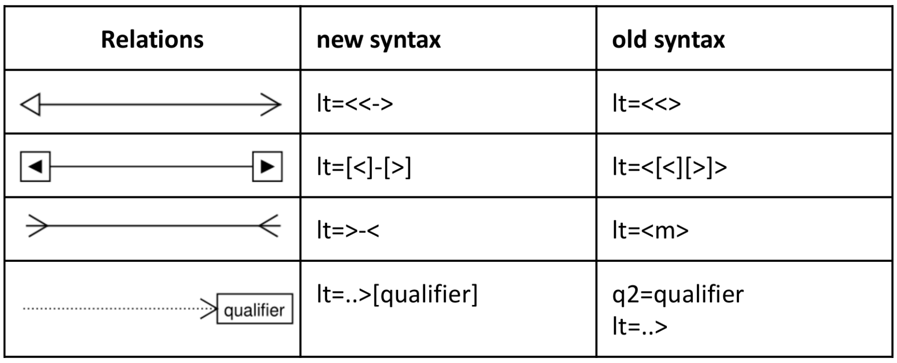
The new architecture enables much more flexible and customizable elements. Due to this customizations, around 50 very specialized elements were reduced to 20 more generic elements.

Relations

As figure 5.4 shows, the syntax to create a relation has been simplified to a generic three-part-line-type.

```
lt=<left-arrow><line-type><right-arrow>
```

- line-type is equal to the line-type described in the **lt=** row of table 5.1
- left-arrow and right-arrow are described in table 5.2



Relations	new syntax	old syntax
	lt=<<->	lt=<<>
	lt=[<]-[>]	lt=<[<][>]>
	lt=>-<	lt=<m>

qualifier	lt=..>[qualifier]	q2=qualifier
		lt=..>

Figure 5.4: Comparison old and new syntax of relations

RegEx Left	**RegEx right**	**Description**
\\[[^\\]]*\\]	\\[[^\\]]*\\]	A box with an optional text in it
\\[[^\\]]*\\]<	>\\[[^\\]]*\\]	like the box but with an arrow pointing to it
<	>	a simple arrow
>	<	an inverted arrow
<<	>>	a closed arrow
<<<	>>>	a filled closed arrow
<<<<	>>>>	a diamond arrow
<<<<<	>>>>>	a filled diamond arrow
\\)	\\(	a half circle
\\(\\)	\\(\\)	a circle
\\(\\+\\)	\\(\\+\\)	a circle with a cross in it
x	x	a diagonally crossed line

Table 5.2: New Relation Line Endings

Figure 5.5 shows a comparison of several more complicated old and new relations.

Other NewGridElements

An overview of all other NewGridElements can be seen in figure 5.6.

Elements which are separate, but behave very similar in many regards share the same background in the figure. So an **UMLDeployment** has the same behavior as an **UMLFrame**, but the UMLFrame moves the first text block to the upper left title.

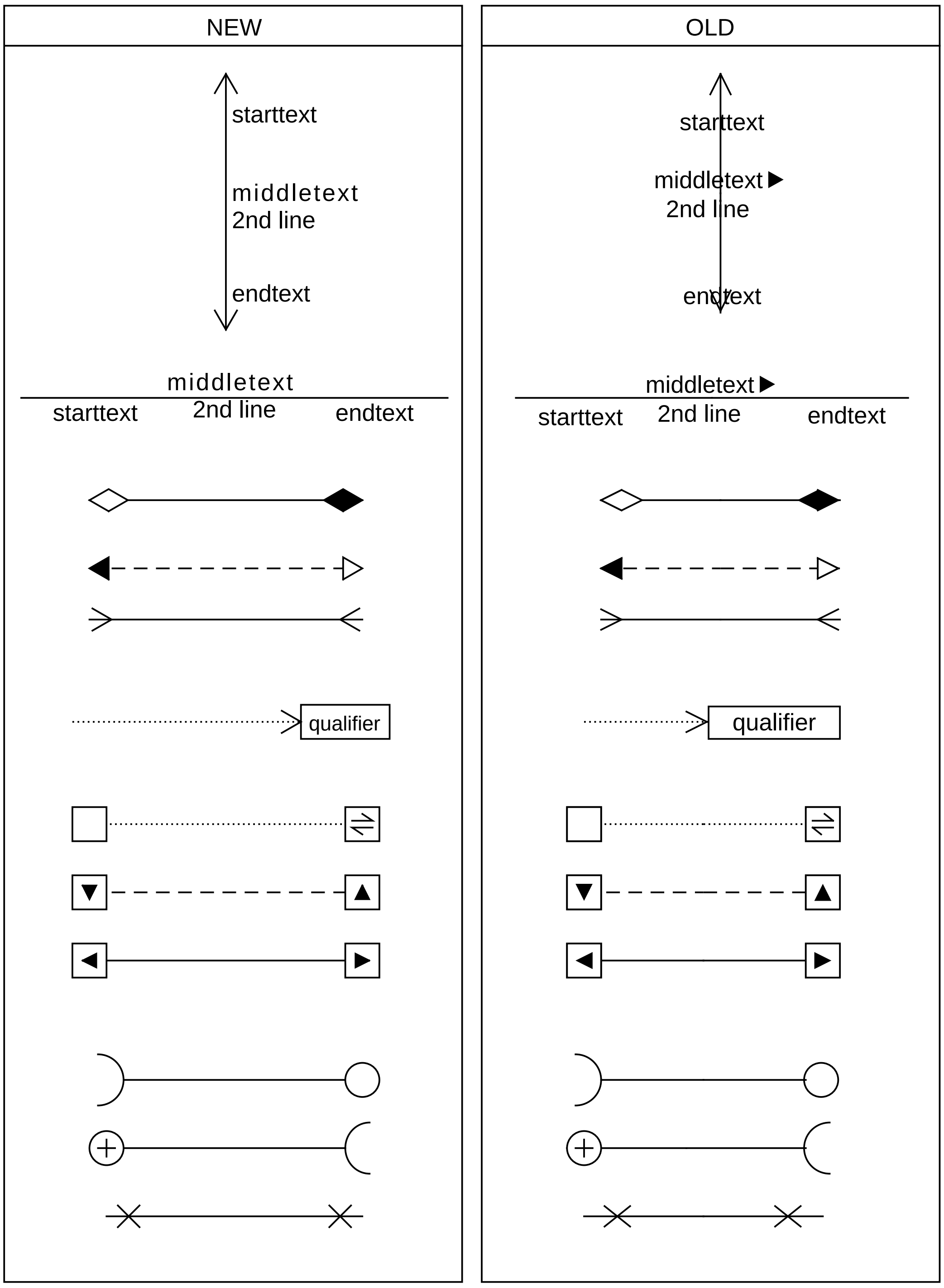


Figure 5.5: Some variants of the new Relation

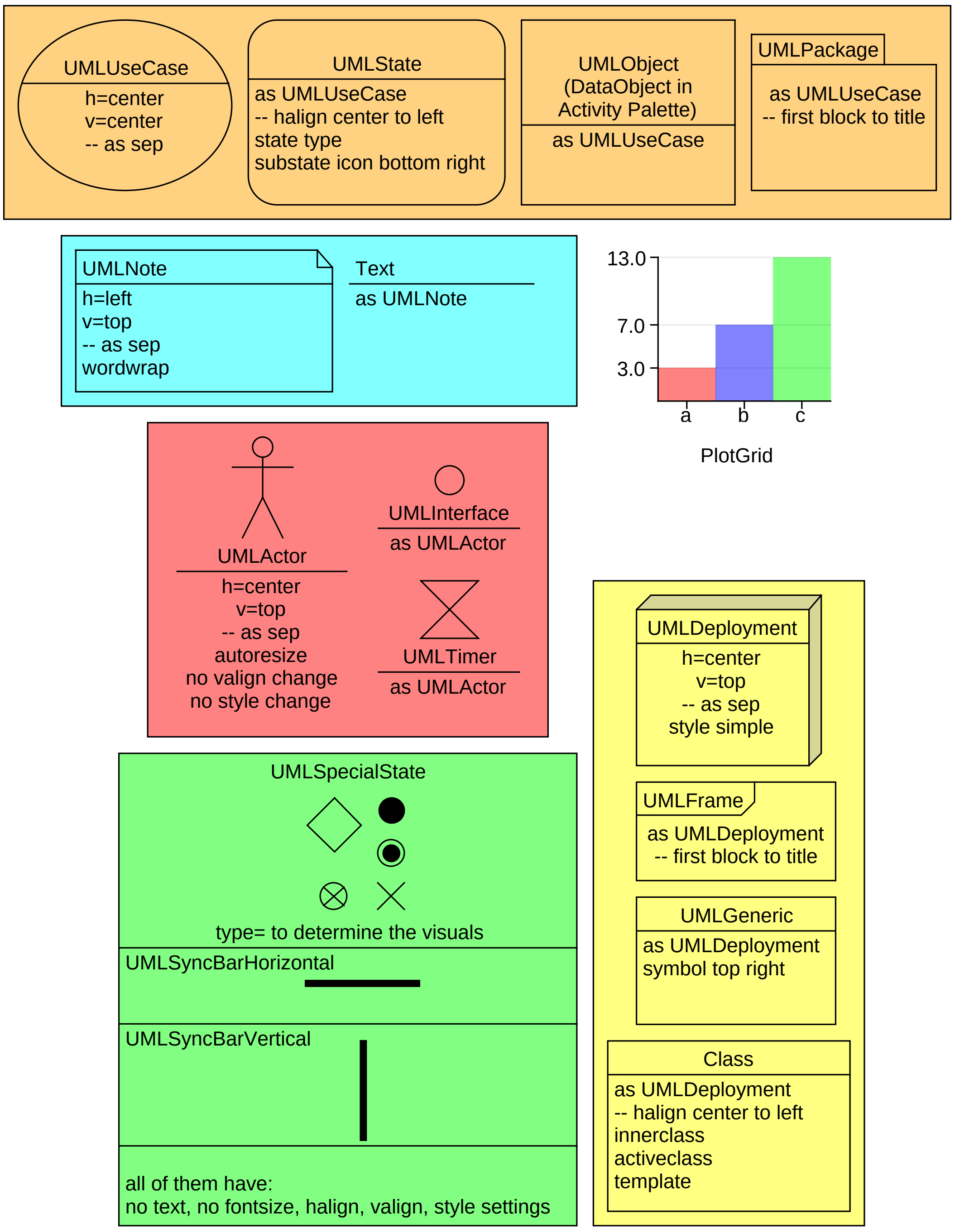


Figure 5.6: Overview of NewGridElements

CHAPTER 6

Web Version Implementation

This section describes the necessary steps to achieve **(M2) Web Version Implementation** from section 1.3. It begins with a short introduction of the general idea of reusing existing Java code with GWT, describes the design goals of the application and shows the implementation steps in detail. The first step is a prototype to ensure overall technical viability of the approach, followed by the realization of several features using iterative changes. Finally the finished web version called UMLetino is presented.

6.1 Reuse Java Code with GWT

As described in section 3.6, the Java Plugin based approach has several disadvantages and failed to work in the past, therefore this time it is planned to run the program in the browser without the need for a plugin.

As the default technology stack of modern browsers is HTML+CSS+JavaScript, the web version should be based on these technologies.

The essence of UMLet is a canvas and functions to decide how to transform text into diagram elements which are drawn on the canvas. Therefore only very small portions of UMLet can be implemented using HTML or CSS and the bulk of the web version must be written in JavaScript.

To avoid maintaining code which does basically the same but is written in a different programming language than Java, the choice of GWT is obvious as it's a widespread and stable web framework which compiles Java to JavaScript (see chapter 3.4) and therefore allows to reuse large portions of the Java codebase.

Missing Java Emulations

As described in 5.3, only parts of the Java SE API are emulated by GWT, but missing parts can be implemented if necessary.

In general this can be a more or less severe disadvantage of choosing GWT. If the project has several 3rd party dependencies, they must be made GWT compatible or replaced by GWT compatible alternatives.

In the case of UMLet the only classes which are used but not emulated are the GUI framework specific AWT classes. To fix this problem, these classes have been replaced with platform independent self written code (see chapter 5.4).

Other than the Java SE API classes, there is one relevant third party library which is used by the shared project: **log4j**, a commonly used Java logging framework[1]

To avoid manual implementation of the missing class, the library **log4j-gwt**[2] is used, which redirects log4j calls to the GWT emulated java.util.logging.Logger.

6.2 UI Design Goals

The design goals of the Web UI are:

- UMLet users should feel familiar when using UMLetino.
- The UI should look like a web application.
- The UI should be very minimalistic and simple.
- As in UMLet browser pop-ups (Window.alert) should be avoided as much as possible.

6.3 Application Menu

According to this design goals, the part which is most likely to change is the application menu.

UMLet uses the same menu structure as many desktop applications (see figure 6.1).

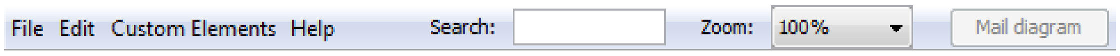


Figure 6.1: UMLet Application Menu

[1] `http://logging.apache.org/log4j/1.2/`
[2] `http://log4j-gwt.sourceforge.net/`

The top space of the application shows broad categories like **File**, **Edit**, **Help** and a click expands the menu to show menu actions which fit into the category (see figure 6.2)

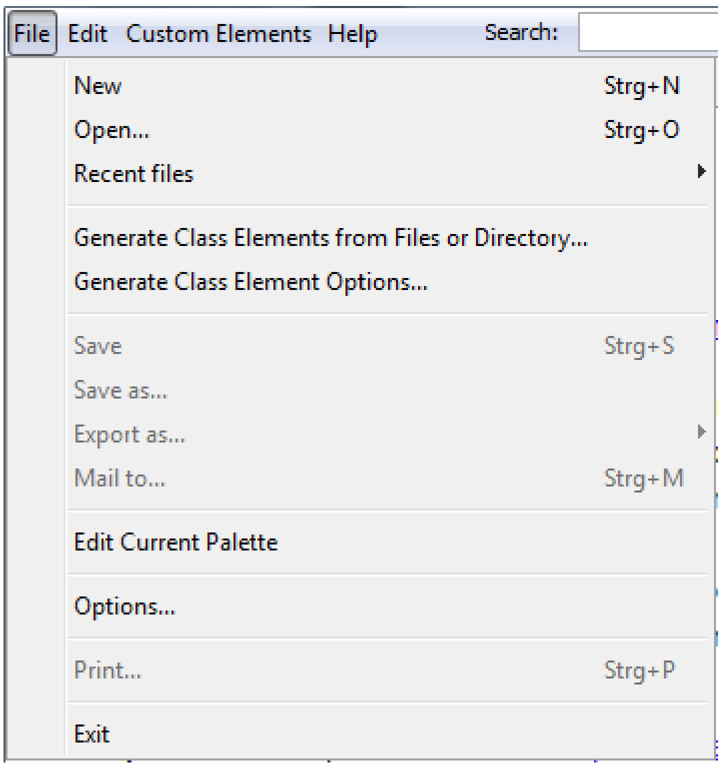


Figure 6.2: UMLet Application Menu expanded

GWT offers some widgets which can reproduce the look and feel of such a menu [3], therefore the first approach was to create a mock of the original UMLet menu using GWT which can be seen in figure 6.3.

After some time of development, it became apparent that, although the menu works, it looks very atypical for a web application.

Therefore the next iteration was to change the menu to a more conventional approach for a web-application, namely a fixed menu on the left side (see figure 6.4). Such a menu also has the advantage of using horizontal space instead of vertical space, which is preferable, as most modern monitors have a 16:9 aspect ratio.

[3]GWT Showcase Menubar: `http://samples.gwtproject.org/samples/Showcase/Showcase.html#!CwMenuBar`

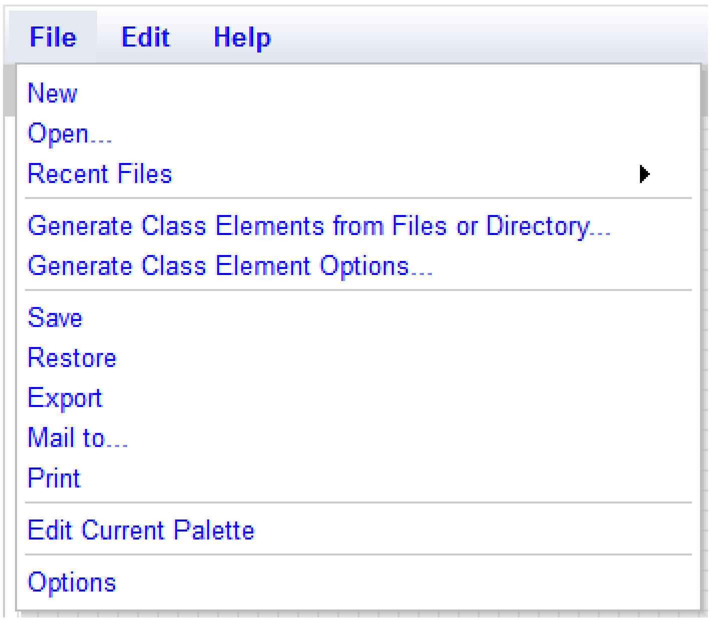


Figure 6.3: UMLet based GWT menu mock

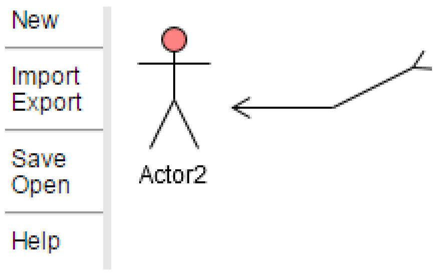


Figure 6.4: UMLetino left menu side first iteration

The next iteration was the removal of the **New** entry (because tabs have been removed) and the removal of the **Open** entry, because stored diagrams are now directly represented as menu items which can be opened by clicking on the name or deleted by clicking on the **X** on the left side of the name.

In the last few iterations, only minor changes were applied, like adding the UMLetino logo and the version, but the simplistic design stayed the same and the final menu can be seen in figure 6.5. In this sample screen shot 3 diagrams have been saved, called **dia1**, **dia2**, **dia3**.

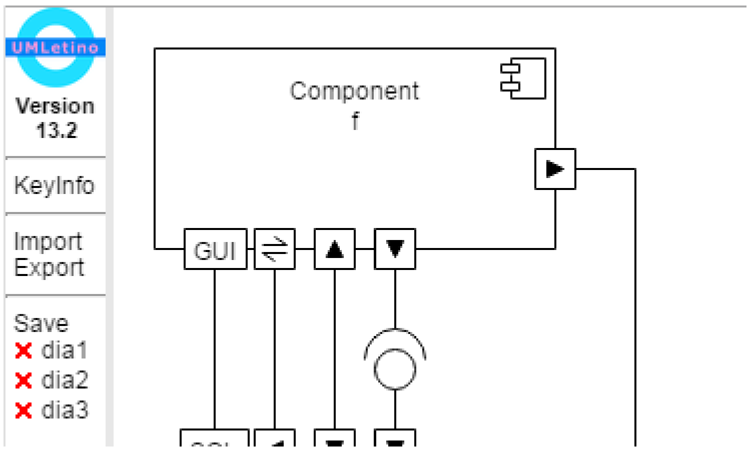


Figure 6.5: UMLet based GWT tab bar mock

6.4 Diagram Tab Bar

UMLet features a tab bar which is located directly under the menu bar (see figure 6.6). Every tab represents a diagram and the user can switch between them and copy elements from one diagram to another one.

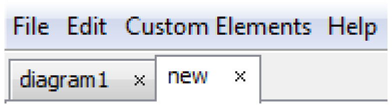


Figure 6.6: UMLet tab bar is located under the menu

As with the menu, the first approach was to rebuild the tab bar using GWT. The result can be seen in figure 6.7

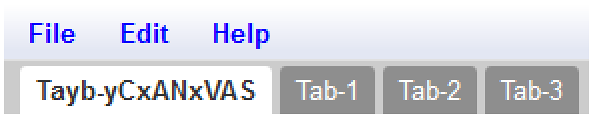


Figure 6.7: UMLet tab bar is located under the menu

As all modern browsers support tabs on their own, it can be confusing to place a tab bar directly under another tab bar, so the idea was to use the browsers tab bar instead of implementing a separate one. Also it wouldn't be possible to prevent users from using browser-tabs, therefore this redundancy could only be cleared by removing the application-specific tab-bar.

The only question concerning the browser tabs was about tab interaction. As mentioned, in UMLet it's possible to copy an element from one tab to another one, therefore this functionality should also be available using browser tabs.

Fortunately this feature came for free after implementing the HTML-Local-Storage based Clipboard. Because both browser tabs work with the same URL, they access the same local-storage and therefore the same clipboard.

Therefore the tab bar has been removed in later versions without a UI component replacing it.

6.5 Palette and Properties

The location of the palette and the properties panel works well for a web application, therefore there were no major changed compared to Standalone UMLet.

One important aspect of the Properties Panel is the auto-completion which can be implemented based on GWTs SuggestBox [4].

Without going into too much detail, some major customizations were necessary to get the correct behavior, because the properties panel has some specific requirements which are not met by a typical suggest-box like being an HTML text-area instead of an HTML input element or showing all suggestions if the shortcut **Ctrl+Space** is used.

Fortunately the SuggestBox code is modular enough to mimic UMLet's properties panel behavior. As an example figure 6.8 shows the suggest-box which is shown if **Ctrl+Space** is pressed on an empty line and 6.9 shows the reduced suggestions if the user starts typing.

6.6 Saving and Opening Diagrams

As mentioned in 3.3, most browsers do not allow the web application to write files to the filesystem. This is a problem, because UMLet diagrams are typically stored in uxf files which are written to the filesystem.

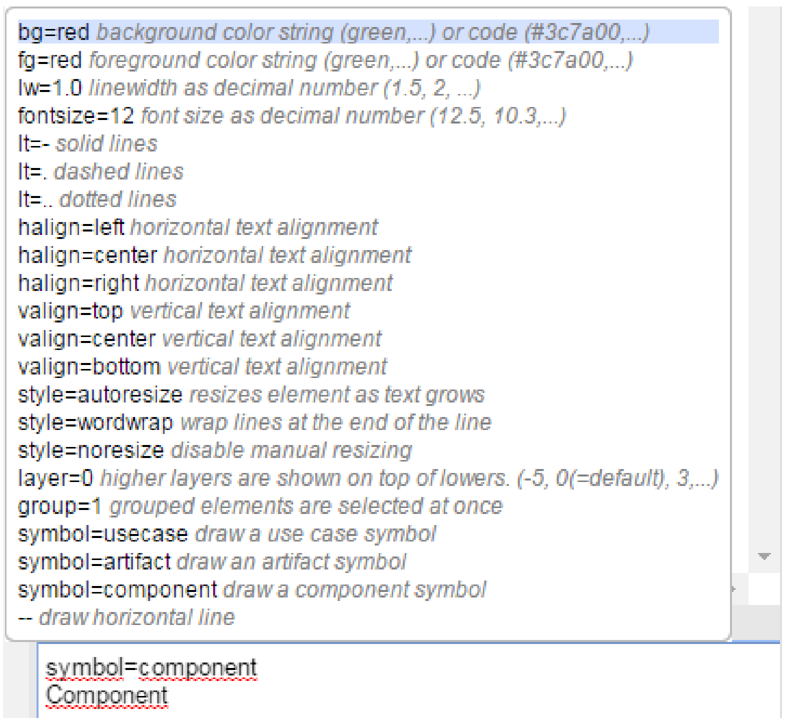
Therefore a storage mechanism must be found which better fits to a web application. This can typically be one of 2 things:

1. Storing diagrams in the local storage of the browser.

2. Storing diagrams remotely on a server.

As currently UMLetino is designed to work offline, even if the user opens it from his file system, the remote storage solution is not implemented. Therefore diagrams are stored as uxf files in the local storage of the browser, which will persist diagrams as long as the user doesn't change the browser or clears his cache. To open a diagram, the user simply clicks on its menu entry (see figure 6.5).

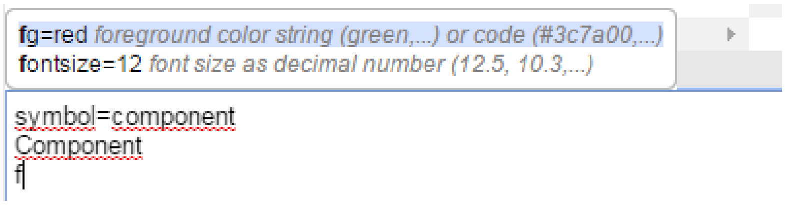
[4]GWT Showcase SuggestBox: `http://samples.gwtproject.org/samples/Showcase/Showcase.html#!CwSuggestBox`



```
bg=red background color string (green,...) or code (#3c7a00,...)
fg=red foreground color string (green,...) or code (#3c7a00,...)
lw=1.0 linewidth as decimal number (1.5, 2, ...)
fontsize=12 font size as decimal number (12.5, 10.3,...)
lt=- solid lines
lt=. dashed lines
lt=.. dotted lines
halign=left horizontal text alignment
halign=center horizontal text alignment
halign=right horizontal text alignment
valign=top vertical text alignment
valign=center vertical text alignment
valign=bottom vertical text alignment
style=autoresize resizes element as text grows
style=wordwrap wrap lines at the end of the line
style=noresize disable manual resizing
layer=0 higher layers are shown on top of lowers. (-5, 0(=default), 3,...)
group=1 grouped elements are selected at once
symbol=usecase draw a use case symbol
symbol=artifact draw an artifact symbol
symbol=component draw a component symbol
-- draw horizontal line
```

```
symbol=component
Component
```

Figure 6.8: UMLetino Auto-completion all suggestions



```
fg=red foreground color string (green,...) or code (#3c7a00,...)
fontsize=12 font size as decimal number (12.5, 10.3,...)
```

```
symbol=component
Component
f
```

Figure 6.9: UMLetino Auto-completion restricted suggestions

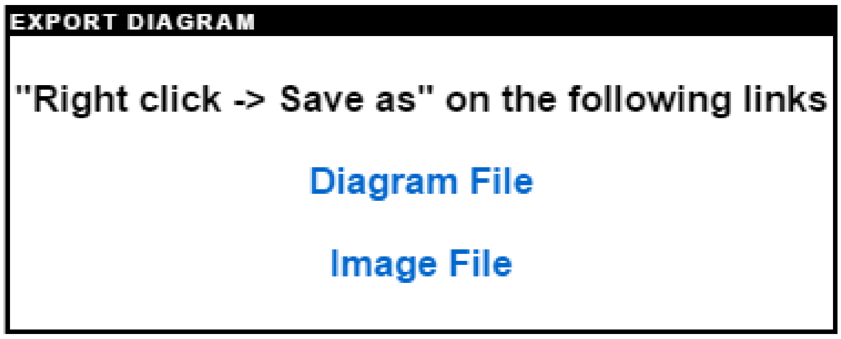
6.7 Importing and Exporting Diagrams

There are several use cases where the local storage based saving mechanism is not sufficient and a separate import/export mechanism is needed:

- Opening UMLetino diagram with UMLet.
- Opening UMLet diagram with UMLetino.
- Sharing diagrams between browsers or with other users.
- Export a diagram as an image file.

All modern browsers support the File API (see chapter 3.3) for reading files into a web application. Therefore the user can click on the menu item **Import** to get a file selection pop-up in which he can choose the diagram file to open. Alternatively he can simply drag the uxf file into the empty diagram and it will import it from the file.

The menu item **Export** allows the user to export a diagram into using different formats. As mentioned earlier browsers typically do not allow a web application to write a file to the filesystem, therefore as a workaround for this issue, the diagram is converted into a data URI which the user can save using **Right Click -> Save as** (see figure 6.10)


EXPORT DIAGRAM

"Right click -> Save as" on the following links

Diagram File

Image File

Figure 6.10: UMLetino Export Overlay

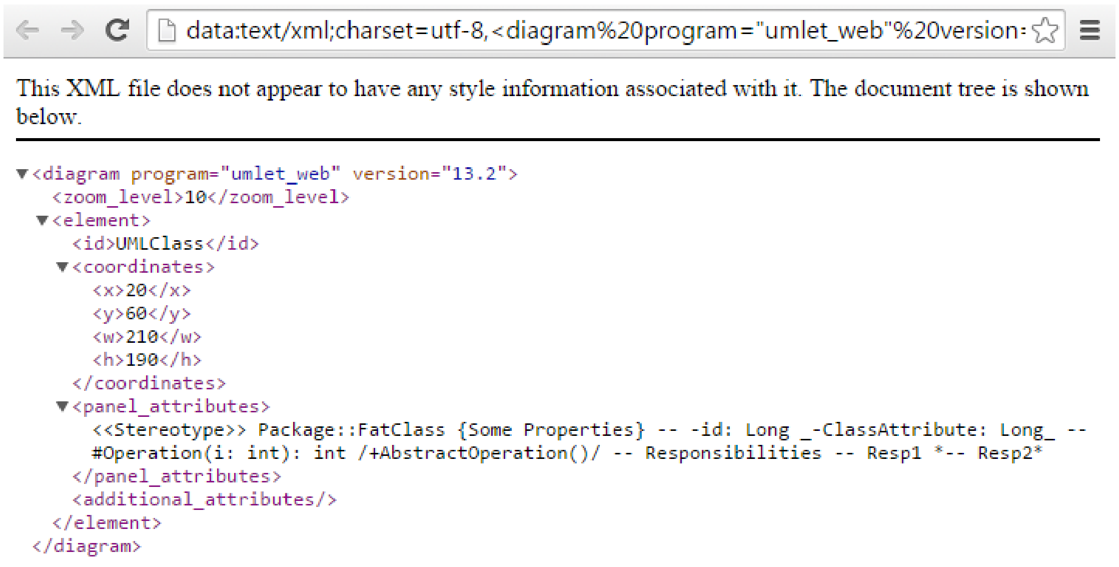
The browser will scan the URL and act depending on the MIME-Type[5] used in the Data URI[6] prefix:

- **data:text/xml;charset=utf-8**: Browser will detect an XML file and show the text content in the browser tab or store the XML file if the user saves it.
- **data:image/png;base64**: Browser will detect a base64 encoded png image file and show it in the browser tab or store the appropriate png file if the user saves it.

Examples how Data URIs look like if the are opened in a browser tab can be seen in figures 6.11 and 6.12

[5]MIME Types are defined in RFC 2045: `https://tools.ietf.org/html/rfc2045` and RFC 2046 `https://tools.ietf.org/html/rfc2046`

[6]Data URIs are defined in RFC 2397: `http://tools.ietf.org/html/rfc2397`


data:text/xml;charset=utf-8,<diagram%20program="umlet_web"%20version

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<diagram program="umlet_web" version="13.2">
   <zoom_level>10</zoom_level>
  ▼<element>
     <id>UMLClass</id>
    ▼<coordinates>
       <x>20</x>
       <y>60</y>
       <w>210</w>
       <h>190</h>
     </coordinates>
    ▼<panel_attributes>
       <<Stereotype>> Package::FatClass {Some Properties} -- -id: Long _-ClassAttribute: Long_ --
       #Operation(i: int): int /+AbstractOperation()/ -- Responsibilities -- Resp1 *-- Resp2*
     </panel_attributes>
     <additional_attributes/>
   </element>
 </diagram>
```

Figure 6.11: Data URI example for UXF file in the Chrome browser

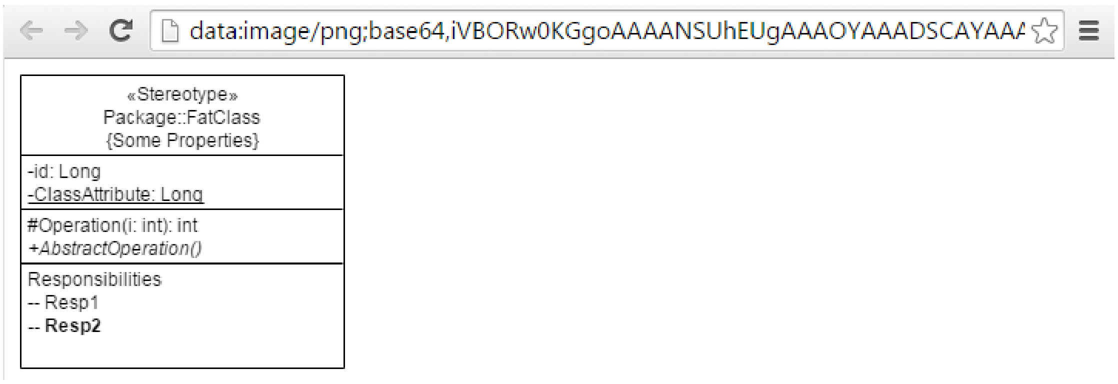


Figure 6.12: Data URI example for PNG file in the Chrome browser

Special characters within the UXF file (like spaces) must be URL-encoded to make them work within a Data URI (see figure 6.11). The browser transforms them back to the original character in the XML view and in the downloaded XML file.

Possible disadvantages of this approach compared to Standalone UMLet:

- **Right Click -> Save as** is not very user friendly. A better way would be opening a file save pop-up.
- limited export options (currently an HTML canvas only supports png. export)

- browser and OS specific differences (e.g., users have reported that iOS limits the File Reader API and therefore the import functionality to image files which is not sufficient as UMLet requires the import of text files).

These issues with IO-operations in general could be solved by implementation of a server side where diagrams can be stored (see chapter 10.3)

6.8 Clipboard

As access to the clipboard of the operating system is currently not supported by most browsers, the local storage of the browser is used as a replacement.

As the local storage is a key-value based string storage, that means there is a reserved key for the clipboard diagram (in uxf format) which gets overwritten with each copy command and read with each paste command.

The user experience is the same for the typical use case of a user working with UMLetino using one browser and one or more browser tabs with UMLetino instances, but there are some minor differences for specific use cases:

- Advantage: The clipboard content stays in the local storage, even if the computer is restarted.
- Disadvantage: User cannot paste the copied elements into a running Standalone UMLet version or into an UMLetino tab which runs in another browser or domain.
- Disadvantage: Standalone UMLet also stores the copied elements as an image file which can then be inserted in several programs like graphics editors (e.g., Adobe Photoshop) or word processors (e.g., MS Word). This is not possible using local storage.

In a future version, the clipboard can be improved by using the W3C Clipboard API (see chapter 10.3) as soon as most browsers support it.

6.9 Keyboard Shortcuts

Because UMLetino is running within browsers which offer keyboard shortcuts themselves, some problems can occur with:

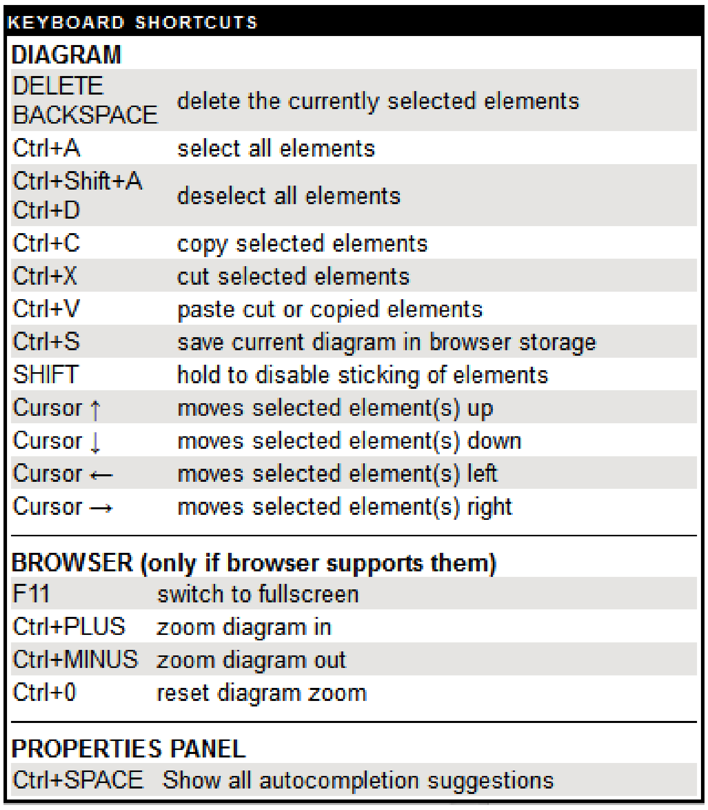
- Overlapping shortcuts (e.g., **Ctrl+C** typically copies the selected text, but in UMLetino diagrams it should copy the selected elements).
- Browser shortcuts which work fine on typical web pages but not on web applications (e.g., **Ctrl+F** for searching is useful on most web pages, but shouldn't really do anything if used in UMLetino.
- Shortcuts which should behave differently based on the focused element (e.g., while the user is within the properties panel, the browser default shortcuts should apply).

To solve these issues in a generic way, UMLetino focuses on retaining the expected behavior using the browser shortcuts wherever it makes sense and only overwrites them if necessary:

- If a shortcut is used within a diagram, all overwritten shortcuts (like **Ctrl+C**) are checked and if a match is found it is executed. If no match is found the default browser shortcut is applied (therefore all typical actions like **close tab** still work).
- In a component which is text based (like the properties) browser shortcuts should not be overwritten but only extended if necessary (e.g., the properties show the auto-completion if **Ctrl+Space** is pressed but do not change the behavior of **Ctrl+C** for copying text into the clipboard).

The only disadvantage of this way to handle shortcuts is that some shortcuts, which do not make sense if applied on a diagram, still can be executed (e.g., **Ctrl+F** still shows the typical search box pop-up of the browser). Although this can be kind of confusing it's better than unintentionally overwriting browser shortcuts which could make sense (it's hard to think about all shortcuts of all different browsers)

There is also a menu item which shows an overlay element with all possible keyboard shortcuts (see figure 6.13).



KEYBOARD SHORTCUTS

DIAGRAM	
DELETE BACKSPACE	delete the currently selected elements
Ctrl+A	select all elements
Ctrl+Shift+A Ctrl+D	deselect all elements
Ctrl+C	copy selected elements
Ctrl+X	cut selected elements
Ctrl+V	paste cut or copied elements
Ctrl+S	save current diagram in browser storage
SHIFT	hold to disable sticking of elements
Cursor ↑	moves selected element(s) up
Cursor ↓	moves selected element(s) down
Cursor ←	moves selected element(s) left
Cursor →	moves selected element(s) right

BROWSER (only if browser supports them)	
F11	switch to fullscreen
Ctrl+PLUS	zoom diagram in
Ctrl+MINUS	zoom diagram out
Ctrl+0	reset diagram zoom

PROPERTIES PANEL	
Ctrl+SPACE	Show all autocompletion suggestions

Figure 6.13: UMLetino Keyboard Shortcuts Overlay

6.10 Zoom

In the current iteration there is no specific zoom implementation, because the browser zoom functionality works fine for UMLetino.

Unfortunately there are some issues with the reliance on browser zoom (see chapter 10.3) :

- Mobile browsers typically don't offer such functionality.
- The menu and the properties get zoom although they should stay the same size.

Therefore in the future a specific zoom layer should be introduced to improve the usability of UMLetino (especially on mobile devices)

6.11 Performance Improvements for HTML5 Canvas

As mentioned in 3.3, JavaScript is typically not as fast as Java.

Furthermore Swing has a build in layering and modularization of the components which should be drawn, but a HTML 5 canvas is just a simple area to draw on using generic drawing-functions.

Therefore Standalone UMLet can apply several performance improvements automatically which must be implemented manually for UMLetino.

Fortunately the structure of the DrawHandler is already designed in a way where a draw-model is built and reused as long as the properties do not change. In the web version this logic is extended to also reuse a canvas as long as nothing changes. Therefore every NewGridElement has its own canvas (stored in the platform specific ComponentGwt class) which is only redrawn from the DrawHandlers draw-model if:

- the properties must be re-parsed (typically if they or the size of the element changes).
- the selection state of the element is changed, which means its foreground color is changed to blue (the fixed foreground color for selected elements).

After this possible redraw of the elements canvas, it gets drawn on the canvas of the Diagram at the position of the element. See figure 6.14.

This enables UMLet to only redraw the parts of the diagram which really change and therefore increase the performance of the application.

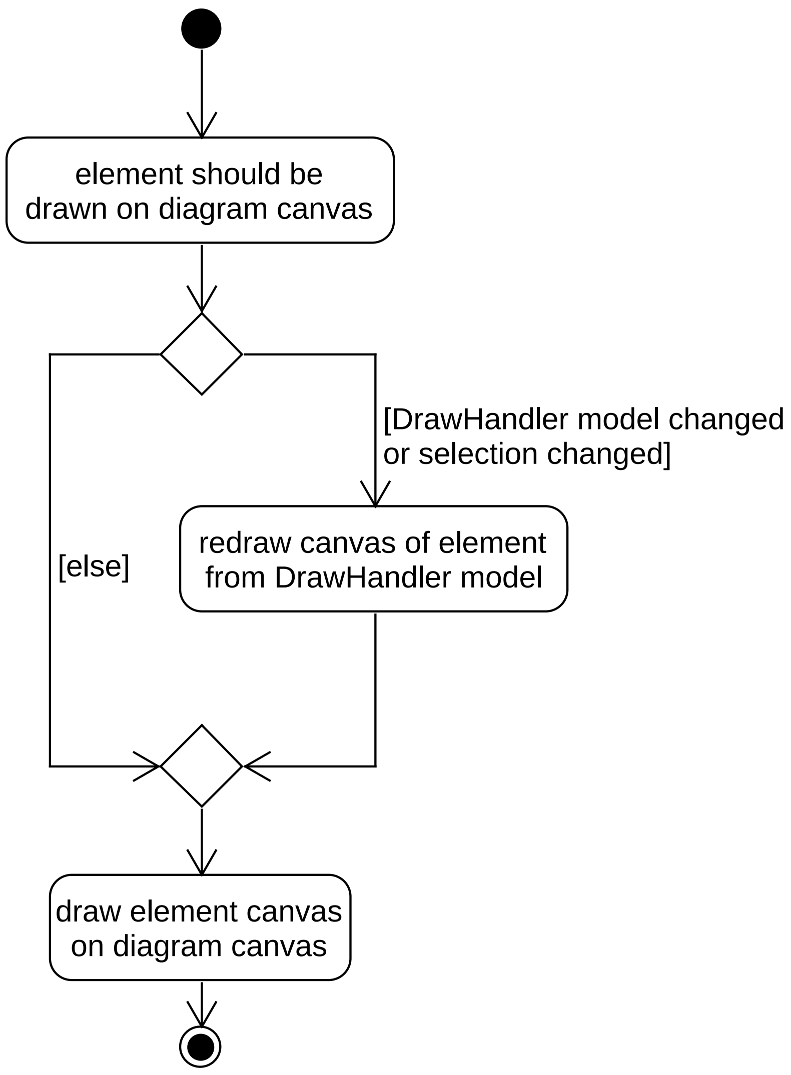

element should be drawn on diagram canvas
[DrawHandler model changed or selection changed]
[else]
redraw canvas of element from DrawHandler model
draw element canvas on diagram canvas

Figure 6.14: Activity when element.drawOn(diagramCanvas) is called

6.12 Major Development Version Snapshots

Although there have been many iterations during UMLetino development, this section will show and shortly describe some major milestones during development.

The first milestone (see figure 6.15) already shows the distribution of most UMLetino building blocks:

- The placement of the diagram, palette, palette chooser and properties panel stayed nearly unchanged up to the final release (although the content of these components only consists of filler elements at this point of development).
- At this stage of development, the diagram elements were simple blue blocks, but it was already possible to move them around within the diagram borders.
- In terms of UI design, the main difference compared to the final version is the mocked menu bar which looked very similar to the UMLet menu bar.
- Another major difference compared to later versions is the tab bar, which will soon be removed, because it's functionality is redundant with the browser tabs.

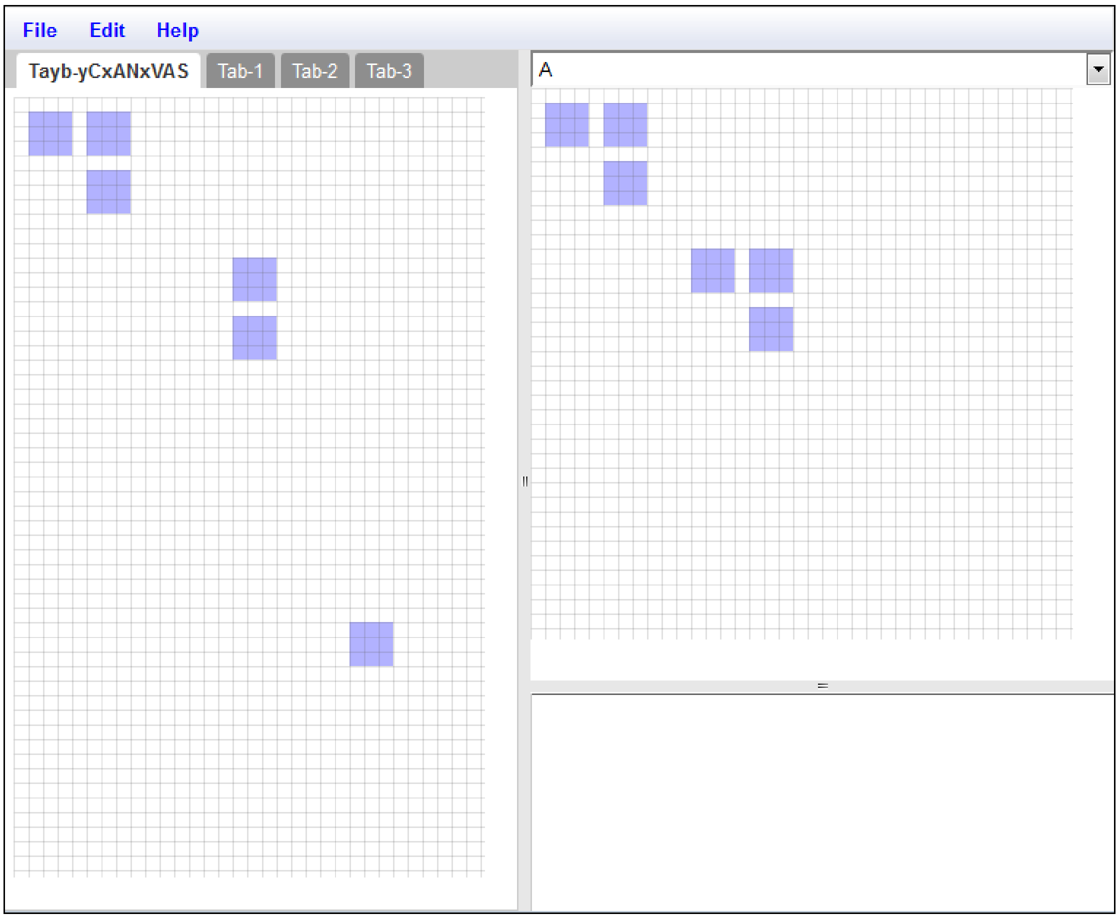
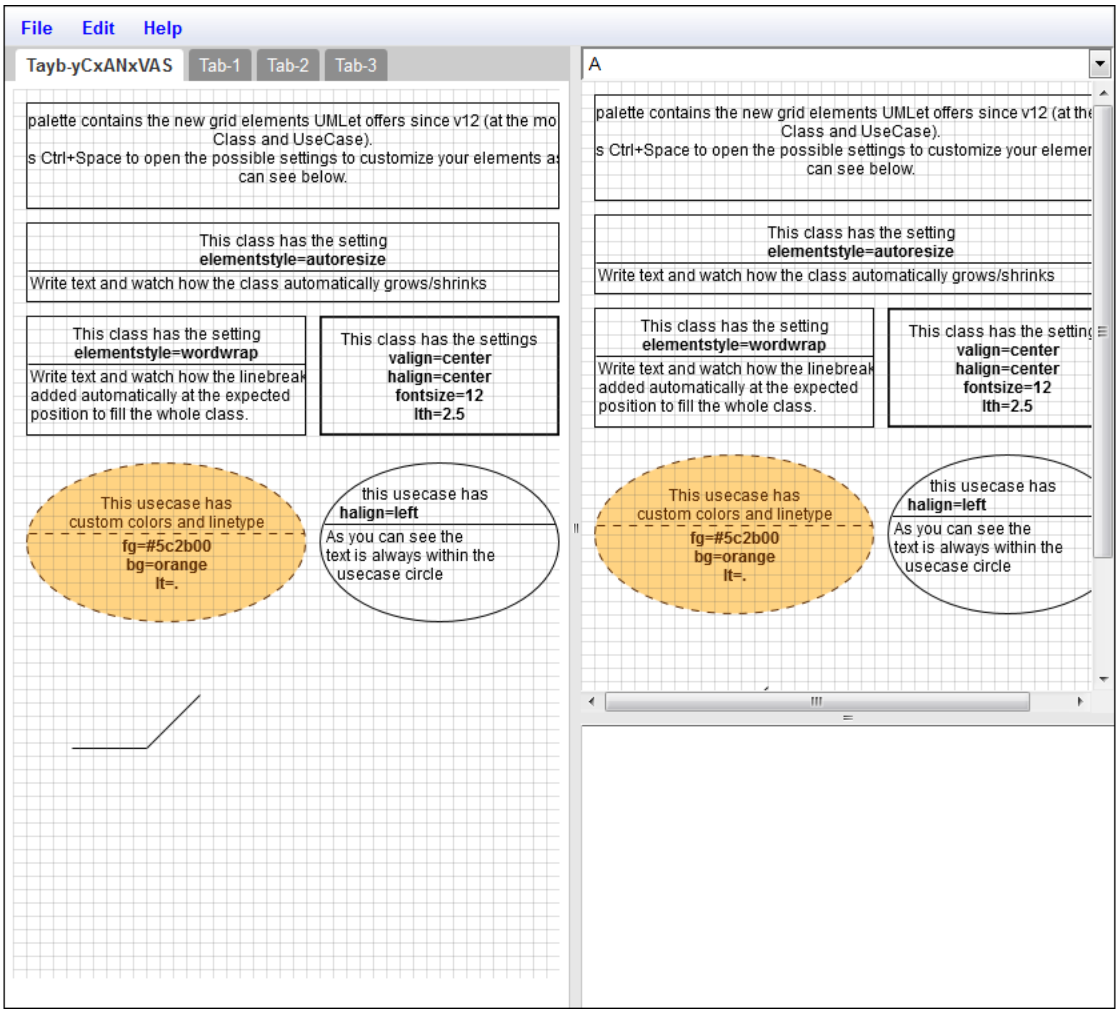


Figure 6.15: UMLetino Web-Application milestone 1

The second milestone (see figure 6.16) was mostly about getting the GridElements to work.

Because the GridElement code was already moved to a separate shared project, the task of getting them to run in GWT was mostly about implementing the relevant interfaces (i.e. the **DrawHandler** implementation for GWT)

Although the elements already worked and were correctly parsed from uxf files, they were read from a constant String holding the uxf file instead of parsing files or opening them from the local storage.



palette contains the new grid elements UMLet offers since v12 (at the mo
Class and UseCase).
s Ctrl+Space to open the possible settings to customize your elements a
can see below.

This class has the setting
elementstyle=autoresize
Write text and watch how the class automatically grows/shrinks

This class has the setting
elementstyle=wordwrap
Write text and watch how the linebreak
added automatically at the expected
position to fill the whole class.

This class has the settings
valign=center
halign=center
fontsize=12
lth=2.5

This usecase has
custom colors and linetype
fg=#5c2b00
bg=orange
lt=.

this usecase has
halign=left
As you can see the
text is always within the
usecase circle

palette contains the new grid elements UMLet offers since v12 (at the
Class and UseCase).
s Ctrl+Space to open the possible settings to customize your elemer
can see below.

This class has the setting
elementstyle=autoresize
Write text and watch how the class automatically grows/shrinks

This class has the setting
elementstyle=wordwrap
Write text and watch how the linebreak
added automatically at the expected
position to fill the whole class.

This class has the setting
valign=center
halign=center
fontsize=12
lth=2.5

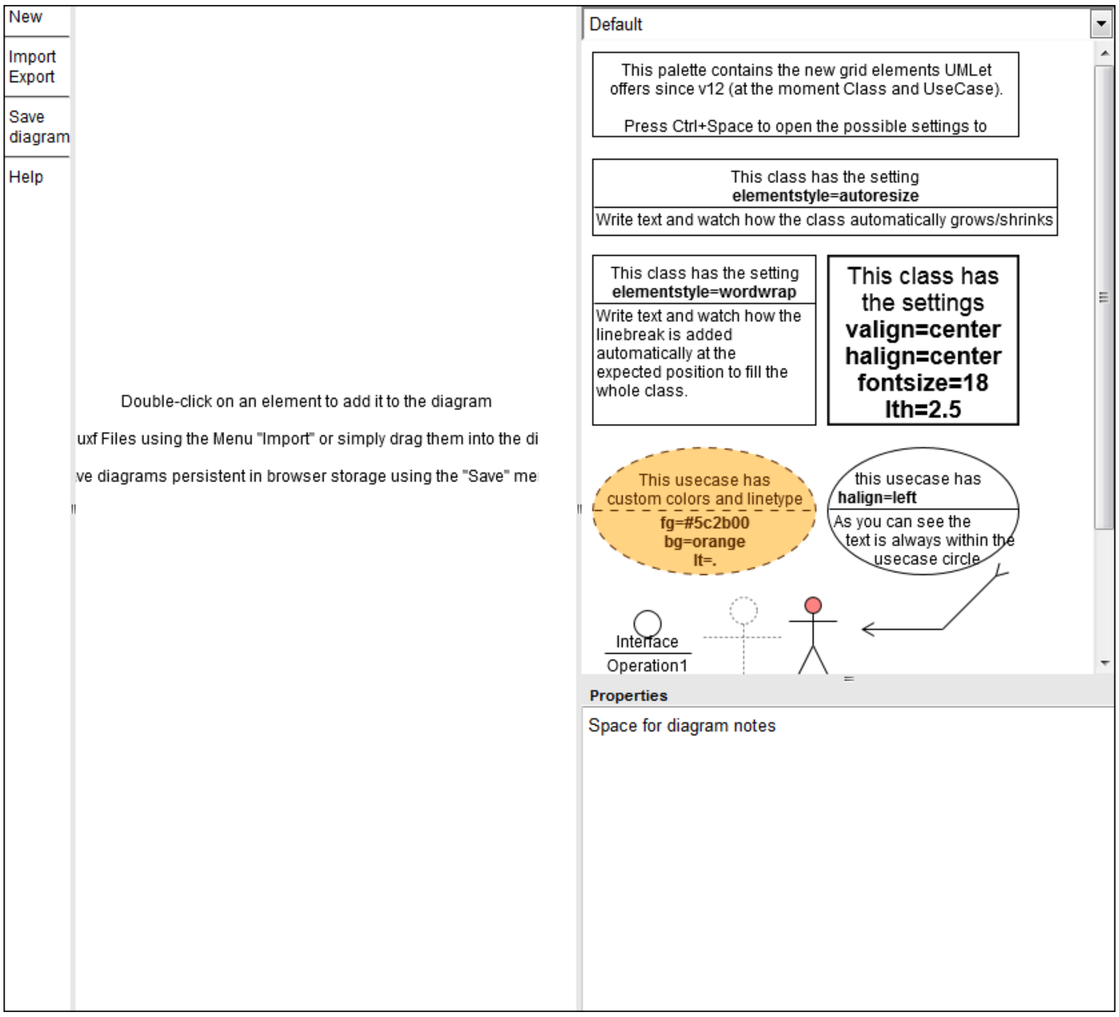
This usecase has
custom colors and linetype
fg=#5c2b00
bg=orange
lt=.

this usecase has
halign=left
As you can see the
text is always within the
usecase circle

Figure 6.16: UMLetino Web-Application milestone 2

The third milestone (see figure 6.17) shows many changes to the UI:

- The menu has moved to the left and changed to a better fitting menu for a web application.
- The tab bar has been removed.
- The empty diagram help-text is shown if no elements are present.
- The properties have a headline **Properties**.
- The grid is no longer shown (it can still be accessed if the **?dev** parameter is added to the URL).



Double-click on an element to add it to the diagram

uxf Files using the Menu "Import" or simply drag them into the di

ve diagrams persistent in browser storage using the "Save" me

This palette contains the new grid elements UMLet offers since v12 (at the moment Class and UseCase).

Press Ctrl+Space to open the possible settings to

This class has the setting **elementstyle=autoresize**

Write text and watch how the class automatically grows/shrinks

This class has the setting **elementstyle=wordwrap**

Write text and watch how the linebreak is added automatically at the expected position to fill the whole class.

This class has the settings **valign=center** **halign=center** **fontsize=18** **lth=2.5**

This usecase has custom colors and linetype **fg=#5c2b00** **bg=orange** **lt=.**

this usecase has **halign=left**

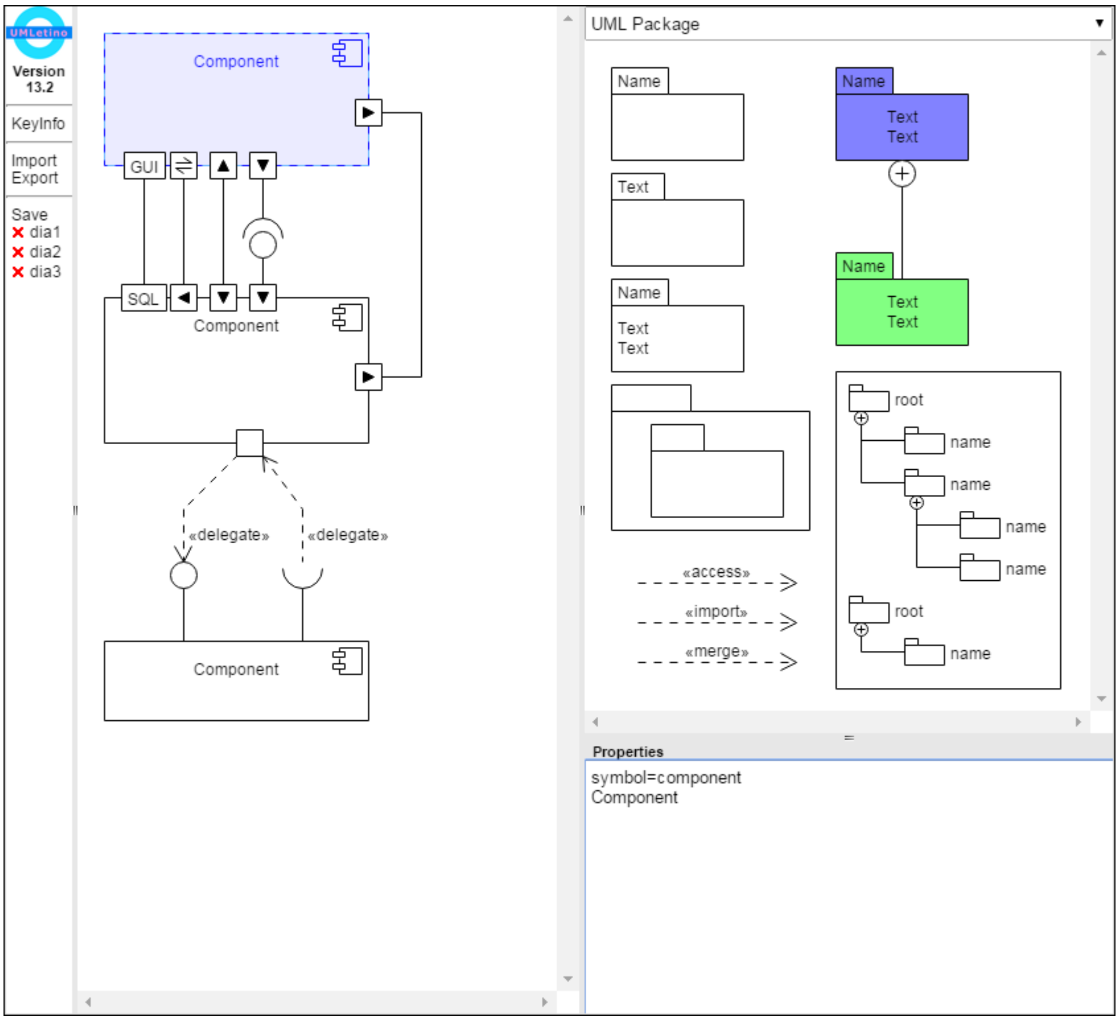
As you can see the text is always within the usecase circle

Space for diagram notes

Figure 6.17: UMLetino Web-Application milestone 3

The final version (see figure 6.18) once more shows several changes to the UI:

- The menu shows the UMLetino logo and the version string at the top left corner.
- The palette is finished and offers the same entries as UMLet (excluding the palettes with OldGridElements which wouldn't work in UMLetino).
- Saved diagrams show a red **X** to delete them.



symbol=component
Component

Figure 6.18: UMLetino Web-Application final

CHAPTER 7

Code Analysis

This section analyzes the success of the chosen reengineering approach in terms of a clean and modular project structure.

1. Findbugs is applied to the codebase before and after the refactoring to see if the refactoring steps have fixed potential errors throughout the codebase.

2. A graph of the project dependencies shows that the desired hierarchy has been successfully implemented (endpoint projects depend on the shared code but not the other way around and endpoint projects are independent from each other).

3. The amount of shared code for all platforms is calculated and compared

7.1 UMLet Eclipse Projects

Before starting with the code analysis, the names of the Eclipse projects are explained.

Baselet

Before the migration started, UMLet only consisted of this one Eclipse project. At the point of the code analysis, this project still contains UMLet standalone and Eclipse plugin specific code, but none of the code which is shared with the web version UMLetino.

The name was chosen at a time when the spin-off project Plotlet was actively maintained and Plotlet specific code was also contained in this project and only extracted during the build using an ant-script. The name should express that the project is the base of UMLet and Plotlet, but as of today, plots have been merged into UMLet as a palette and Plotlet no longer exists, therefore the names will be changed back to UMLet during the migration to Maven (see chapter 8.2).

BaseletElements

This project contains the code which is shared by all platforms and therefore used by both other projects Baselet and BaseletGWT. The project is not bound to any specific GUI library and the only external dependencies are JUnit for tests and Log4j for logging.

BaseletGWT

The BaseletGWT project contains the GWT specific code for the web version UMLetino.

7.2 Findbugs Analysis

The following code analysis has been made using **Eclipse Luna Service Release 1 (4.4.1), Build id: 20140925-1800**.

UMLet v12 refers to the following code

- **GitHub Path:** https://github.com/umlet/umlet/releases/tag/2013-02-19_UMLet_v12
- **Projects:** Baselet (the project was not split up at this time)

UMLet modularized

- **GitHub Path:** https://github.com/umlet/umlet/commit/ac827fbef47a7cb98df123e4e3028cf79ba34d1f
- **Revision:** ac827fbef47a7cb98df123e4e3028cf79ba34d1f
- **Date:** 2014-09-11 10:41
- **Projects:** Baselet, BaseletElements, BaseletGWT

Findbugs Settings

This statistics have been created using the **Eclipse FindBugs Plugin v3.0.0.20140706-2cfb468** [1].

The plugin has been used with default settings except that the **Minimum rank to report** was set to **20** instead of the default value of 15, to make sure less severe issues are also reported.

UMLet v12 Results

- 234 violations overall.
- 5 of them are in the category **Scariest** and have Rank 1.
- 1 of them is in the category **Scary** and has Rank 8.

[1] http://findbugs.sourceforge.net/

- 3 of them are in the category **Troubling** with 2x Rank 14 and 1x Rank 11.
- The other 225 issues are in the category **Of Concern**.

UMLet modularized Results

- 7 violations overall, all in the category **Of Concern**.
- 4 of them are about potentially dangerous non-short-circuit logic (all of them are part of the old Relation classes which will be removed at some point in the future and therefore not that important to fix).
- 1 issue is about a potential error in a compareTo() method of the java to class diagram generator (should be verified and fixed).
- 1 issue is about string concatenation using + in a loop in the Sequence all in one diagram (only a performance issue).
- 1 issue is about writing a static field from an instance method (necessary because of the way the eclipse plugin works).

Conclusion

Most issues have been fixed in the new version and many potential and some actual bugs have been fixed in the process.

7.3 Lines of Code

The Plugin which has been used to calculate the Lines of Code is **Eclipse Plugin CodePro Analytix v7.1.0.r37x201109091147** [2]

Unfortunately the plugin has not been actively developed since 2011, but it still works mostly without issues on new Eclipse versions.

UMLet v12 Lines of Code

- 22,688 Total (all in Baselet)

UMLet modularized Lines of Code (see figure 7.1)

- 21,419 in Baselet
- 8,915 in BaseletElements
- 3,135 in BaseletGWT
- 33,469 Total

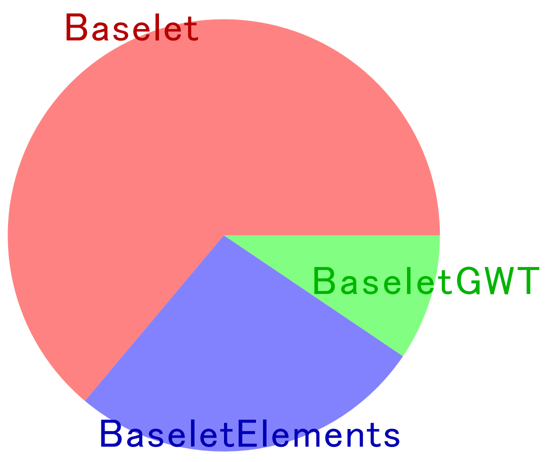


Figure 7.1: Pie Chart of Lines of Code per Project

The 21,419 lines in the Baselet project can be roughly split up into several components (this is only a rough estimation, because the code is old and partially distributed over the whole program through if/else blocks):

Baselet components (see figure 7.2):

- 2,646 lines for the **All in One** Diagrams.
- 1,366 lines for the **Custom Elements**.
- 5,601 lines for the other now **Deprecated OldGridElements**.
- 931 lines for the **Java to Class Diagram Generation**.
- 1,325 lines for the **Eclipse Plugin** integration.
- 9,550 miscellaneous **Swing** specific and/or glue code.

About 5,601 lines of code in the Baselet project are necessary to keep the deprecated elements working (that means every old element except the all in one and custom elements which have not been migrated yet)

[2] `https://developers.google.com/java-dev-tools/codepro/doc/`

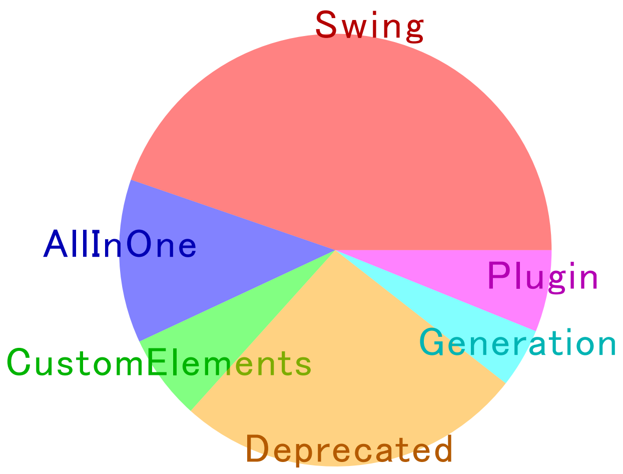


Figure 7.2: Pie Chart of Lines of Code per Baselet Component

Portion of Shared Code

As intended, the GWT project acts as a relatively thin wrapper around the shared functionality of the program. This can be shown as the GWT version has only 3,135 (26%) lines of code while the shared code has 8,915 (74%) lines of code (see figure 7.3).

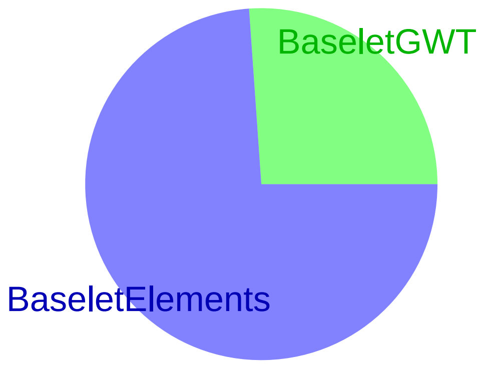


Figure 7.3: Pie Chart of Lines of Code for UMLetino

In comparison, the standalone version has 70,6% specific code and only 29,4% shared code (see figure 7.4).

Even if only the 9,550 lines of swing specific code are used for the calculation, there are still less than 50% of the code in the shared project. One reason for this is the old and historically grown code base with its complex and verbose Swing code.

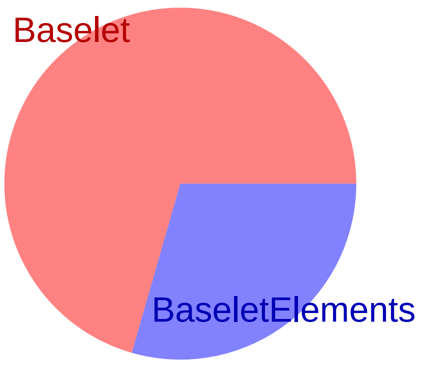


Figure 7.4: Pie Chart of Lines of Code for Standalone UMLet

Listener Refactoring

One component which is currently platform specific, but could be moved in large parts to the shared BaseletElement project is the listener related code.

Standalone UMLet uses a complex listener hierarchy (see figure 7.5) with overlapping and partially redundant Listeners which are attached to the Diagram and to each GridElement. Some of them even have specific subclasses of the GridElementListener like the old Relation element.

UMLetino uses a much simpler approach with only one generic diagram listener. If the mouse is clicked, it is calculated which GridElement is the recipient of the event (based on the mouse position and the layer and size of the elements on that position) and the platform independent event-handling method of the GridElement is called.

This simpler approach could also work in the Standalone version and would make it possible to move the bulk of the event-handling code into the shared BaseletElements project, increasing code reuse and consistency of the user experience over several platforms (see chapter 10.2).

7.4 Project and Package Dependencies

The Eclipse version and compared projects are the same as in 7.2, with the exception that **UMLet modularized** includes some further adaptions necessary to clean up the dependency graph.

Project Dependencies

Figure 7.6 shows the dependencies of the new project structure; the **BaseletGWT** and **Baselet** projects depend on **BaseletElements** but not the other way around. Also **Baselet** and **Baselet-**

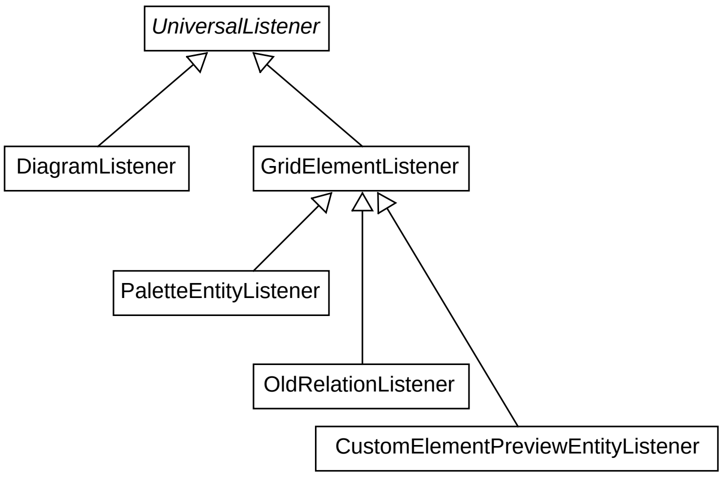


Figure 7.5: UMLet Listener Hierarchy

GWT are independent.

Basic libraries which are used in all projects:

- **rt.jar:** The Java SE Runtime.
- **junit.jar:** JUnit, a commonly used Java testing framework [3].
- **log4j.jar:** Log4j, a commonly used Java logging framework[4].

Baselet specific libraries:

- **org.eclipse.jdt.core.jar:** The Java compiler from the Eclipse project which is used to compile CustomElements.
- **org.eclipse.*.jar:** The other org.eclipse.* packages are only relevant for the Eclipse Plugin, but not used in Standalone UMLet.
- **commons-io.jar** Used for Wild card-handling in Batch-mode.
- **bcel.jar, javaparser.jar:** Used for java-class analysis to generate class diagrams.
- **autocomplete.jar, rtextsyntaxtextarea.jar:** Used for a colorized properties- and custom-elements-panel and its auto-completion.
- **mailapi.jar:** Used to share UMLet diagrams per mail.

[3]`http://junit.org/`
[4]`http://logging.apache.org/log4j/1.2/`

- **jlibeps.jar, batik*.jar, itextpdf.jar:** Used for EPS, SVG and PDF export.

BaseletGWT specific libraries:

- **gwt-user.jar:** The GWT specific classes.
- **lib-gwt-file.jar:** An adapter to configure log4j in GWT.

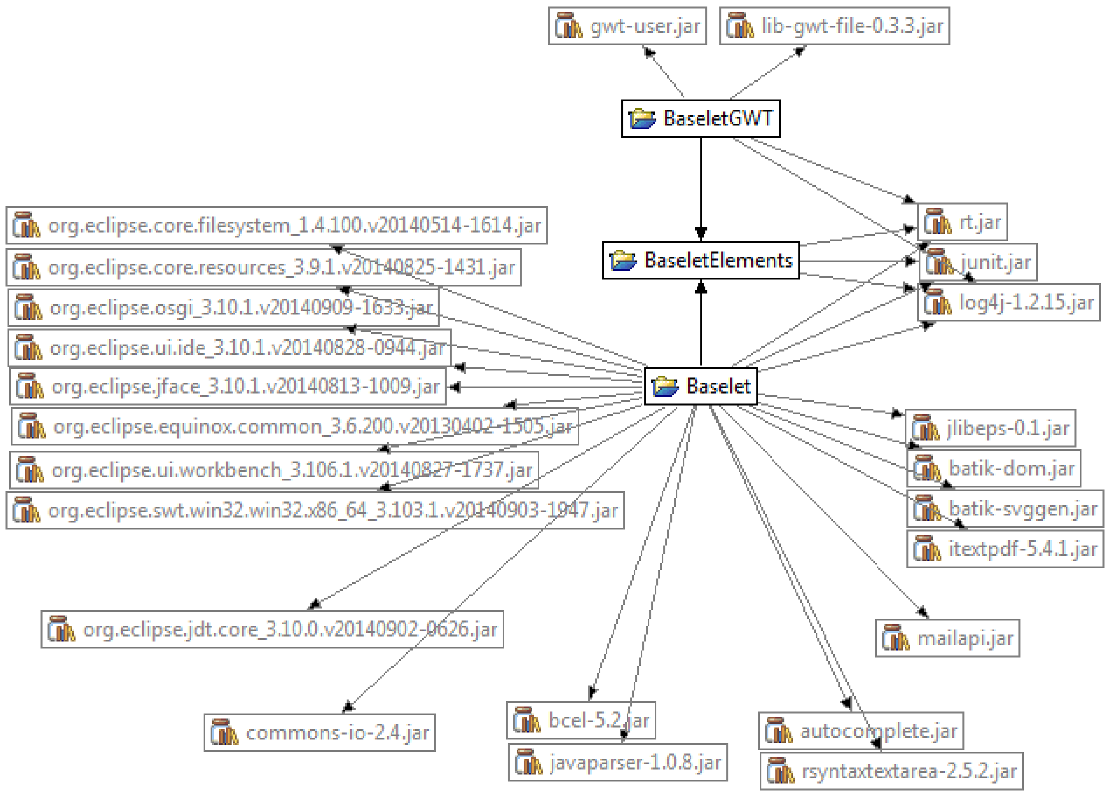

gwt-user.jar
lib-gwt-file-0.3.3.jar
BaseletGWT
BaseletElements
Baselet
rt.jar
junit.jar
log4j-1.2.15.jar
org.eclipse.core.filesystem_1.4.100.v20140514-1614.jar
org.eclipse.core.resources_3.9.1.v20140825-1431.jar
org.eclipse.osgi_3.10.1.v20140909-1633.jar
org.eclipse.ui.ide_3.10.1.v20140828-0944.jar
org.eclipse.jface_3.10.1.v20140813-1009.jar
org.eclipse.equinox.common_3.6.200.v20130402-1505.jar
org.eclipse.ui.workbench_3.106.1.v20140827-1737.jar
org.eclipse.swt.win32.win32.x86_64_3.103.1.v20140903-1947.jar
jlibeps-0.1.jar
batik-dom.jar
batik-svggen.jar
itextpdf-5.4.1.jar
org.eclipse.jdt.core_3.10.0.v20140902-0626.jar
mailapi.jar
commons-io-2.4.jar
bcel-5.2.jar
javaparser-1.0.8.jar
autocomplete.jar
rsyntaxtextarea-2.5.2.jar

Figure 7.6: Dependencies between the 3 UMLet projects and external libraries

As conclusion, the project dependency structure has no cycles and is relatively simple. One possible improvement could be made by extracting the Eclipse-Plugin specific part into its own project, because it's already completely separated from the rest of the code (see chapter 7.8).

This separation should be approached after the switch to a Build and Dependency Management Tool (see chapter 8.1), because handling multiple projects and their dependencies in Eclipse without such tools can be complicated.

Package Dependencies

The package organization of an application reflects the applications structure [55].

The Common-Reuse Principle (CRP) says that *"The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all."* [60] Therefore package design is an important part of Software Engineering.

There are many ways to create a visual representation of the package structure and relationships. One approach for visualization are Package Surface Blueprints which are described by Ducasse et al. [27] in more detail.

In the case of the package dependency analysis for UMLet's projects, the Eclipse Plugin **STAN IDE v2.1.2.v20140325** [5] has been used, as it generated readable and hierarchical diagrams which clearly show the package relationships.

Cyclic dependencies

According to Laval and Ducasse [55], *"A package cycle is a strong coupling between multiple packages that prevents the developer from separating these packages."*

Therefore it is often desirable to detect such cycles and refactor the codebase to remove cyclic dependencies. The goal is to get a clear hierarchical package dependency tree which shows the layers of the sub-components of a project.

The detection of such cycles gets much easier with tool support which either detect and report such cycles or generate a graph which visualize them, e.g., by using red lines as the **STAN** Eclipse Plugin does.

UMLet v12

Figure 7.7 shows that the packages of the old project are tightly coupled for the most part.

This tight coupling has multiple reasons:

- A Main-class which holds all of the relevant program state (currently active diagram, palettes, ...) and does many things (opening diagrams, saving diagrams, displaying notifications, ...).
- Grown, old codebase which has been changed by many developers. Therefore no clear concept of which part of the program should do what.

UMLet modularized - Baselet project

The current version of the Baselet project (see figure 7.8) shows some improvements:

- The java class diagram generator package (previously **umlet.language** package, now **baselet.generator**) is now cycle free.
- The Eclipse Plugin specific classes are not longer referenced by other classes.
- The overall count of cycles has been reduced.

[5]STAN (Structure Analysis for Java): `general/license.html`

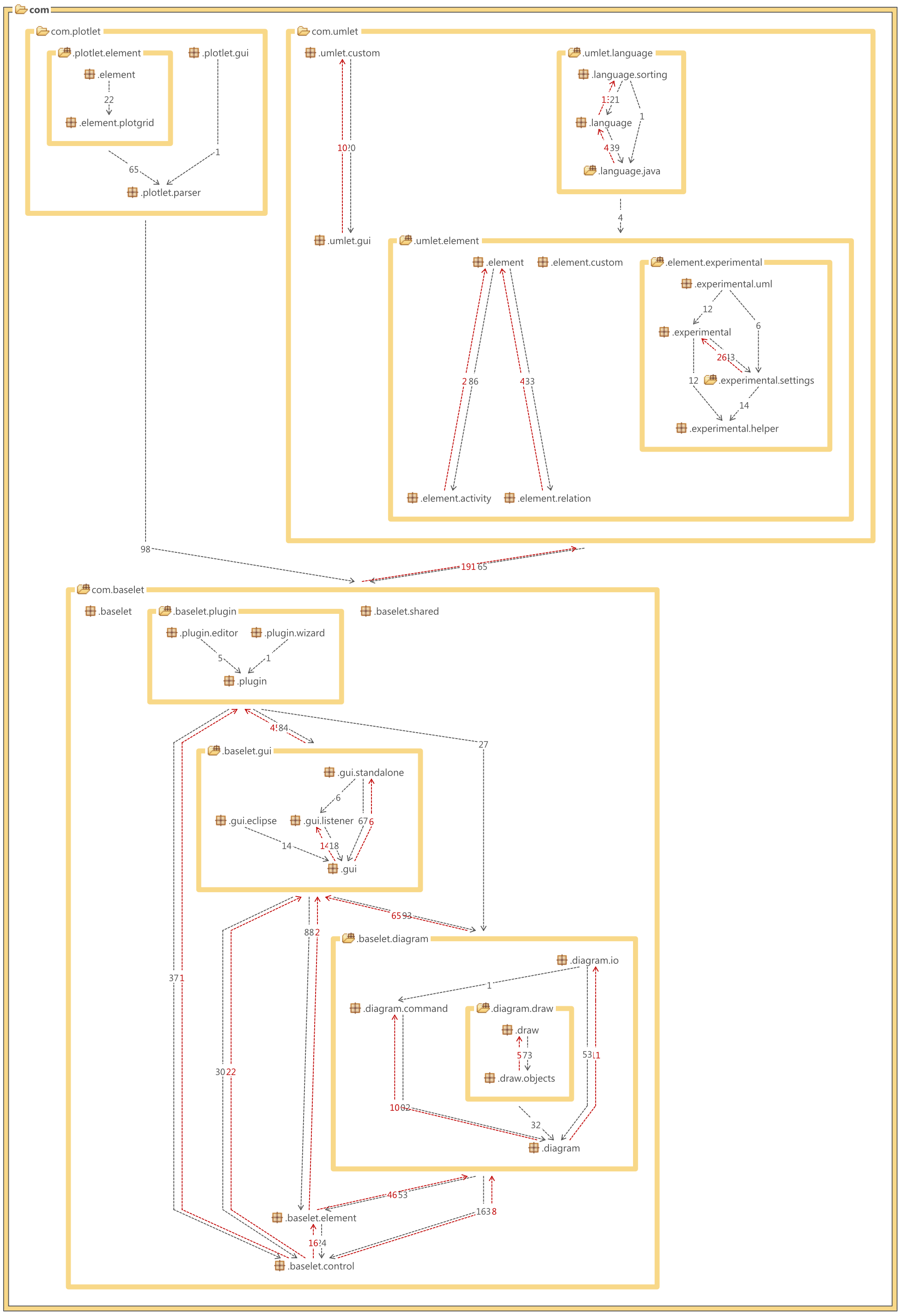


Figure 7.7: Dependencies between packages of Baselet v12 (some minor cycle free sub-packages are not extended for better visibility)

- The old elements are encapsulated in their own package.

There are still remaining issues, like the Main class still does too much and should be split up in several smaller classes. Also there are several cyclic connections between **baselet.diagram** and **baselet.gui**. These components would be easier to understand if they would have a hierarchical call order.

UMLet modularized - BaseletElements project

The dependencies of this project are cycle-free (see figure 7.9), although the basic package structure is very similar to the Baselet project.

There is a **baselet.element** package which contains everything which is element specific, including the parsing logic, facets and the concrete element classes.

The **baselet.diagram.draw** package contains classes to draw the elements, **baselet.control** contains common classes like enumerations, constants, generic data-types like Line, Rectangle, ColorOwn, . . .

Due to this clear top-down-structure, it should be quite easy to understand the basic interaction of the shared program components and to write tests for them, as there are no complex cross-references distributed over many classes.

UMLet modularized - BaseletGWT project

This project is also cycle free (see figure 7.10) and relatively small, compared to the other 2 projects. The main package is **com.baselet.client** which is based on the GWT convention that classes which will be compiled to JavaScript should be placed in a **client** package.

If future releases should support server-side-logic, it will be placed in **com.baselet.server**, while shared classes will be placed in **com.baselet.shared**.

To avoid too strong coupling, the view components have been decoupled by using interfaces (in **com.baselet.view.interfaces**).

Conclusion

The GWT version which depends on BaseletElements and BaseletGWT is cycle free and clearly structured. Therefore they should be easy to understand for new developers.

On the other hand, the Standalone version still has most code in the Baselet project which contains several cycles and unnecessary tight coupling. This should be refactored in a future release.

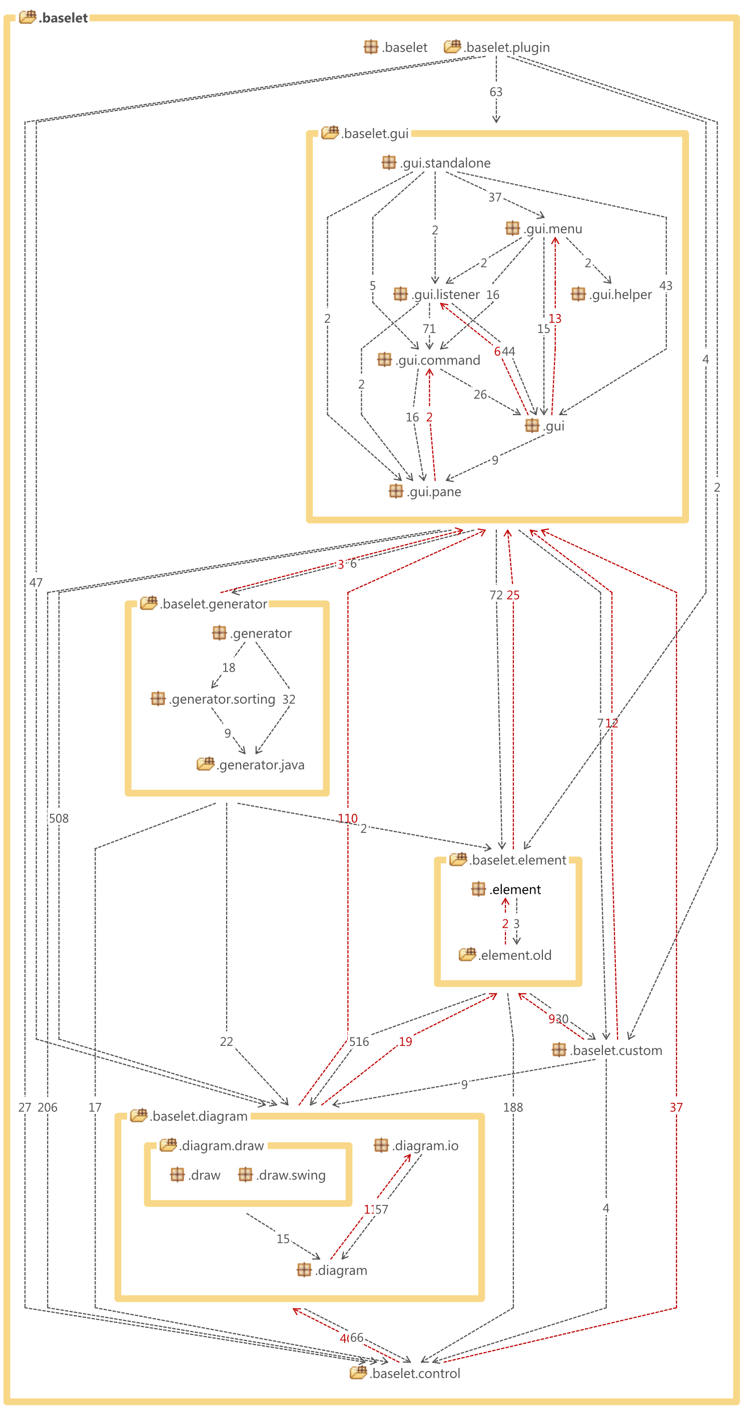


Figure 7.8: Dependencies between packages of Baselet after the modularization (some minor cycle free sub-packages are not extended for better visibility)

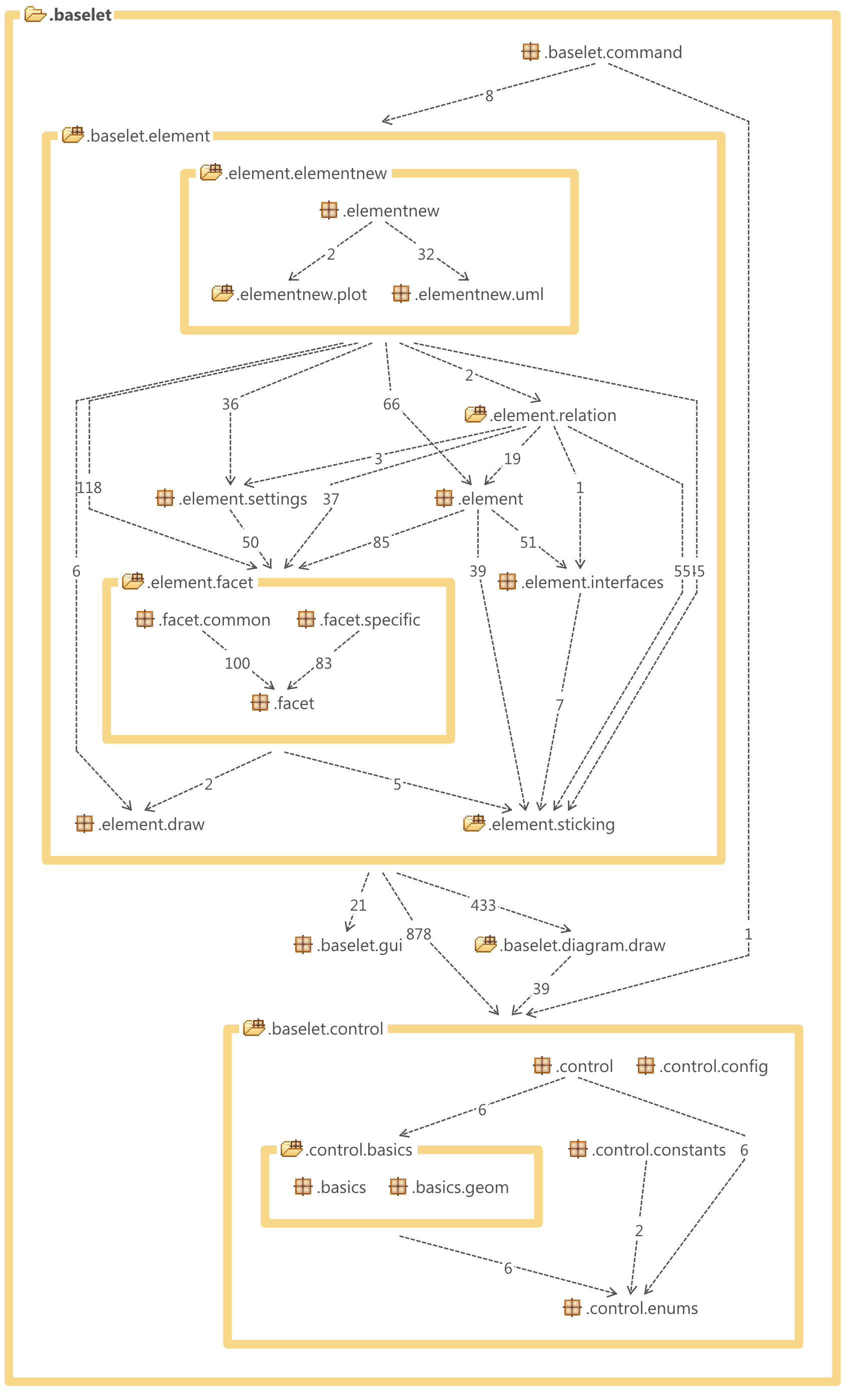


Figure 7.9: Dependencies between packages of BaseletElements (some minor cycle free sub-packages are not extended for better visibility)

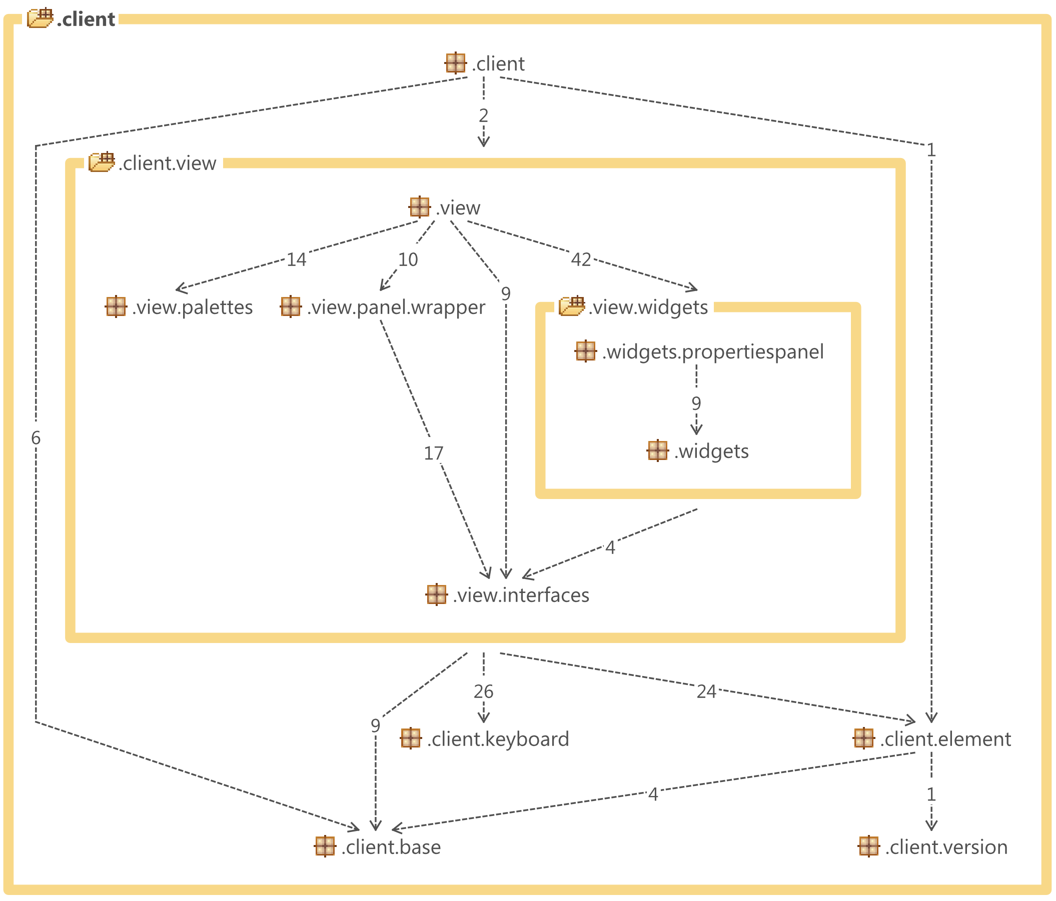


Figure 7.10: Dependencies between packages of BaseletGWT

CHAPTER 8
Maven Build

This chapter documents the migration to the build tool Apache Maven and describes the advantages of the new build system.

This is intentionally done after the code analysis, because it's an independent additional step after the migration, which changes many basic structures of UMLet's codebase (project names, package names, directory structure, . . .), which may compromise the results of the code analysis, which is focused on the modularization and the web version of UMLet.

8.1 Improving Build Structure and Management

At the moment UMLet's internal project dependencies (Baselet, BaseletElements, BaseletGWT) are managed by Eclipse, while external dependencies are simply put into a **lib** directory. There is a **libinfo.txt** file which documents the license, the URL and the usage of every external library. The build procedure is partially automated using Apache Ant build files.

Unfortunately this approach has several issues:

1. The Baselet project mixes Standalone and Eclipse plugin code. If standalone code calls Eclipse plugin code, a runtime error happen, because of missing Eclipse libraries.
2. Runtime dependencies and test dependencies (e.g., JUnit) are mixed.
3. Dependency handling is tied to the Eclipse way of classpath handling and will not work with other IDEs.
4. Building and developing UMLet needs Eclipse plugin dev tools and Eclipse GWT plugin.
5. Main code is mixed with test code and generated code
6. Ant-files can be hard to read and understand, especially for new developers.

These issues can be solved by migrating to a build management tool like Apache Maven:

1. Multi-module projects are possible, which enable fine grained modules to avoid unallowed code references (e.g. Baselet can be split up into 3 modules as seen later)

2. Dependencies have a scope. For example JUnit is typically only used for test code, but not for shipped code. Maven allows to set the scope to test to avoid inclusion of the dependency to the build artifact.

3. Dependency handling is standardized and independent from Eclipse. IDEs provide plugins to transform the maven projects to IDE specific projects and configuration.

4. Building and developing UMLet is possible without any plugins. Eclipse dependencies are now handled by Maven, Eclipse Plugin packaging and GWT transpiler are triggered by Maven plugins. This enables full build automation and continuous integration using a build server.

5. Source code is separated into src/main/java (code which is shipped with the application) and src/test/java directory (code only used for tests). Generated code is typically located in the target directory and not checked into the code repository, therefore developers cannot accidentally change generated code anymore.

6. Maven is widely used and its convention over configuration approach helps new developers to quickly understand the build process and the dependencies and structure of the projects. Standardized Maven plugins improve the build process by running JUnit tests, checking code coverage, running findbugs analysis and much more on every build.

8.2 Migration to a Multi-Module Maven Project

Due to the multi-platform approach of UMLet, the codebase and its dependencies are complex. With Maven it's possible to create a multi-module project with one parent. Every time the parent is built, all of its modules are built automatically.

Figure 8.1 shows the layered multi-module structure. Modules are only allowed to use modules which are located below them in the graph, which results in a simple hierarchical structure without cycles.

The modules can be described as follows:

- **umlet-parent** is the parent project which contains all other projects. Its pom.xml defines basic settings such as the license, java version, common maven plugins and their configuration and more.

- **umlet-elements** is the largest module which contains the shared codebase (previously called BaseletElements)

- **umlet-res** contains shared resources such as palettes, icons, images and more. Currently the web version packages its own resources (i.e. its palettes) and does not depend on this module.
- **umlet-gwt** represents the code which is needed for UMLetino, the GWT based web version of UMLet.
- **umlet-swing** contains the classes which are shared between UMLet standalone and the Eclipse plugin, which is the second largest module, because both endpoints use the same Swing codebase for the most part.
- **umlet-standalone** is a small module which contains the standalone specific endpoint code, which is basically the main method and the argument handling for the batch mode.
- **umlet-eclipse-plugin-deps** is a necessary helper module which collects and packages the dependencies which are needed by the Eclipse plugin. This is necessary to bridge the Maven dependency handling to the OSGi dependency handling which is used by Eclipse plugins.
- **com.umlet.plugin** is the endpoint for the Eclipse plugin and contains its platform specific code. It has to comply with certain restrictions due to the way how Eclipse plugins are handled and therefore uses source-dependencies for UMLet dependencies.

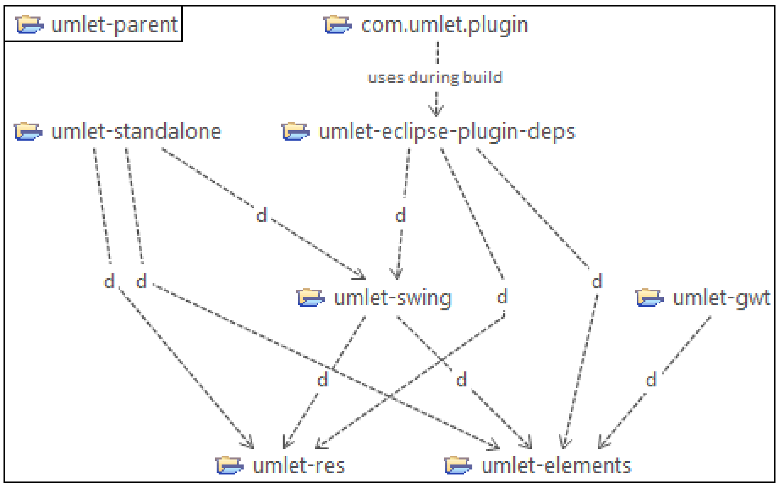


Figure 8.1: Maven Dependency Structure

In addition to the cleaner UMLet module dependencies, figure 8.2 shows that external dependencies are also clearly separated between the different endpoint projects. For example all the Eclipse jar files are only used by the Eclipse plugin, and **commons-io-2.4.jar** is only used by the standalone version for batch argument parsing. Note that the CodePro Analytix tools show the source folder reference for the Eclipse plugin, which is required to bridge maven with osgi dependency handling, as a dependency cycle.

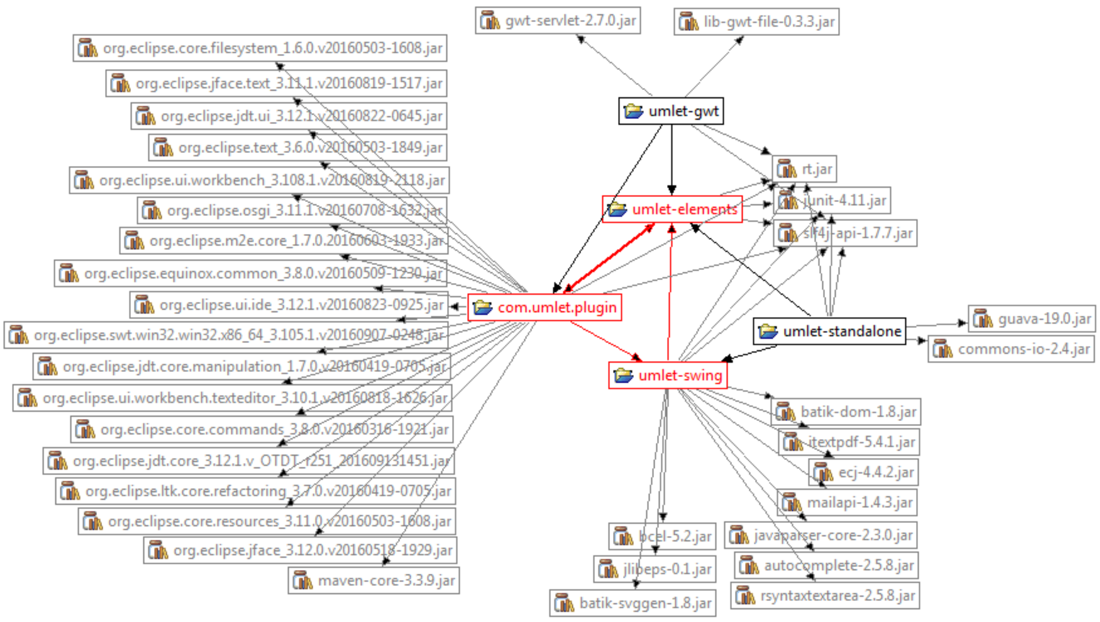

gwt-servlet-2.7.0.jar
lib-gwt-file-0.3.3.jar
org.eclipse.core.filesystem_1.6.0.v20160503-1608.jar
org.eclipse.jface.text_3.11.1.v20160819-1517.jar
org.eclipse.jdt.ui_3.12.1.v20160822-0645.jar
org.eclipse.text_3.6.0.v20160503-1849.jar
org.eclipse.ui.workbench_3.108.1.v20160819-2118.jar
org.eclipse.osgi_3.11.1.v20160708-1632.jar
org.eclipse.m2e.core_1.7.0.20160603-1933.jar
org.eclipse.equinox.common_3.8.0.v20160509-1230.jar
org.eclipse.ui.ide_3.12.1.v20160823-0925.jar
org.eclipse.swt.win32.win32.x86_64_3.105.1.v20160907-0248.jar
org.eclipse.jdt.core.manipulation_1.7.0.v20160419-0705.jar
org.eclipse.ui.workbench.texteditor_3.10.1.v20160818-1626.jar
org.eclipse.core.commands_3.8.0.v20160316-1921.jar
org.eclipse.jdt.core_3.12.1.v_OTDT_r251_201609131451.jar
org.eclipse.ltk.core.refactoring_3.7.0.v20160419-0705.jar
org.eclipse.core.resources_3.11.0.v20160503-1608.jar
org.eclipse.jface_3.12.0.v20160518-1929.jar
maven-core-3.3.9.jar
umlet-gwt
umlet-elements
com.umlet.plugin
umlet-standalone
umlet-swing
rt.jar
junit-4.11.jar
slf4j-api-1.7.7.jar
guava-19.0.jar
commons-io-2.4.jar
batik-dom-1.8.jar
itextpdf-5.4.1.jar
ecj-4.4.2.jar
mailapi-1.4.3.jar
bcel-5.2.jar
javaparser-core-2.3.0.jar
jlibeps-0.1.jar
autocomplete-2.5.8.jar
batik-svggen-1.8.jar
rsyntaxtextarea-2.5.8.jar

Figure 8.2: Maven Dependency Structure

8.3 Build Artifacts

The build has been streamlined and automated for all modules. A single **mvn clean install** builds the whole project and composes the required modules for each platform. Previously the ant build script extracted several parts of the monolithic codebase and composed two output artifacts as shown in figure 8.3 (the figure ignores UMLetino to keep the graphics simple).

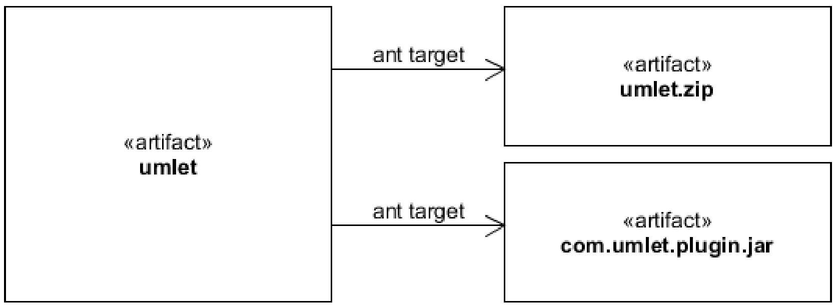


Figure 8.3: Ant script builds output artifacts from monolithic codebase

With the migration to Maven, the projects have been separated into small modules, which are simply composed during the build to create output artifacts for all platforms (figure 8.4).

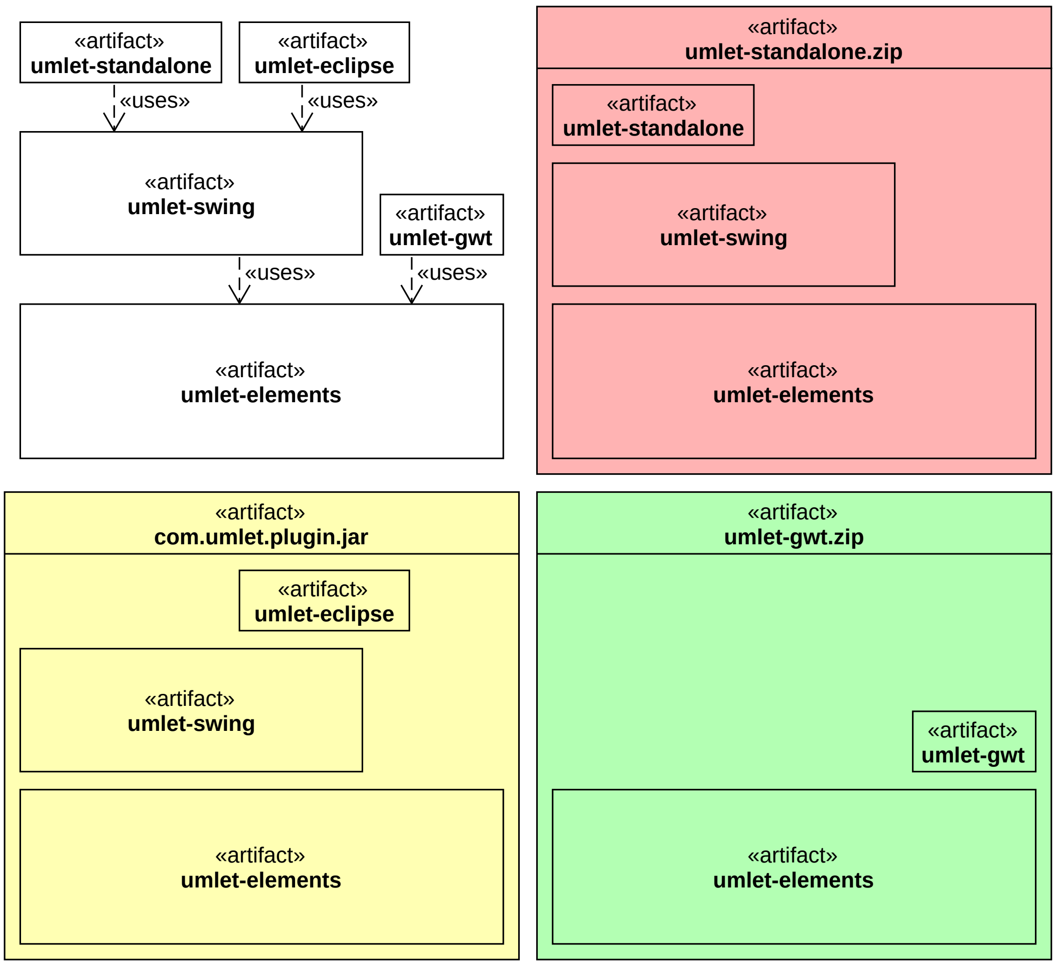


Figure 8.4: Maven modules and how they are composed for each platform (red: UMLet standalone, yellow: UMLet Eclipse Plugin, green: UMLetino)

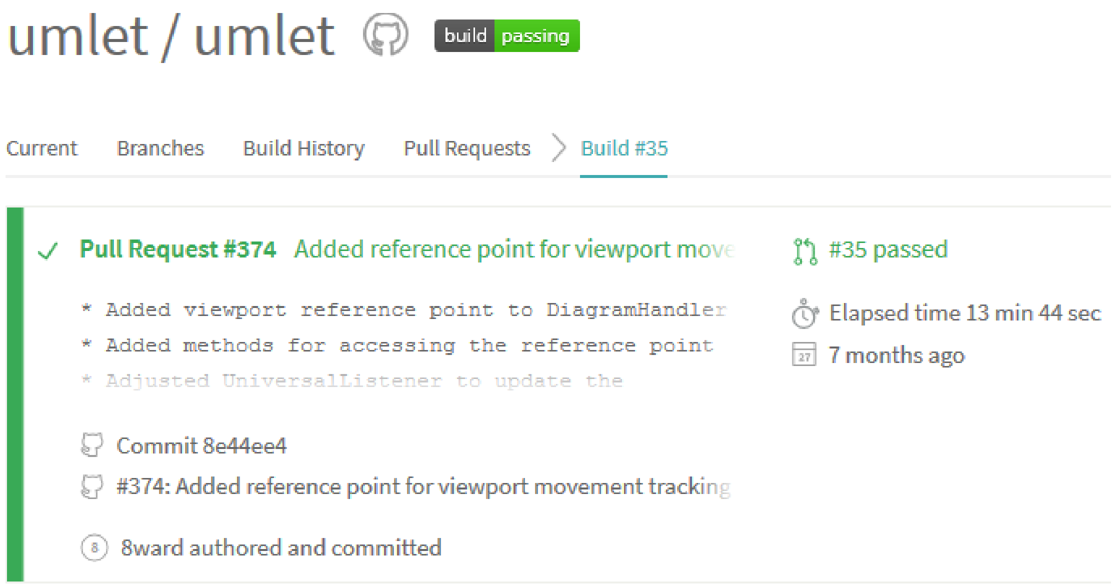
8.4 Continuous Integration with Travis

With the migration to maven, the build is now standardized and decoupled from local Eclipse dependencies. To make sure that the build is possible on an independent system and does not break due to a commit, the UMLet project is now also using the Travis continuous integration server.

Travis automatically builds the project for every commit and therefore shows what effect a commit will have on the build and overall code quality, even before the commit has been merged into the codebase (see figure 8.5). The GitHub integration also marks every commit within the GitHub issue tracker with a green check (if the build succeeds) or red X (if it fails) (see figure 8.6).

A high level of code quality can be guaranteed, because Travis runs all JUnit tests and a Findbugs analysis during the build. If a commit would break tests or violate Findbugs rules, the build will break and the developer has the chance to fix the problem and update the pull request accordingly. This procedure can be repeated until the commits in the pull request are marked green by Travis and later merged by the developer.

A continuous integration server can also be used to make release builds which are automatically uploaded and distributed to the users. In terms of maintenance, it guarantees that a piece of software is actually working and can be compiled, which is important to limit the effects of software aging.



Pull Request #374 Added reference point for viewport move

```
* Added viewport reference point to DiagramHandler
* Added methods for accessing the reference point
* Adjusted UniversalListener to update the
```

Commit 8e44ee4

#374: Added reference point for viewport movement tracking

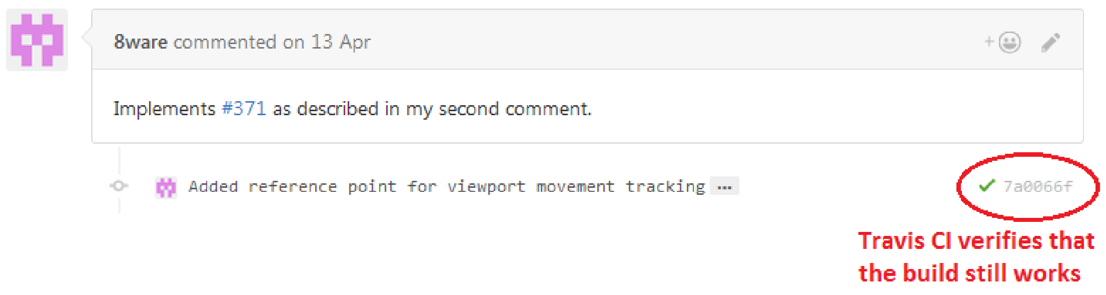
8ward authored and committed

#35 passed

Elapsed time 13 min 44 sec

7 months ago

Figure 8.5: Travis UI shows an UMLet build


8ware commented on 13 Apr

Implements #371 as described in my second comment.

```
Added reference point for viewport movement tracking ...
```

✓ 7a0066f

Figure 8.6: Travis GitHub integration shows if UMLet will build after merging the commit

CHAPTER 9

Discussion of Results

At first, the research questions from section 1.2 are reviewed to show how they have been answered. Afterwards, the successful implementation of the previously defined milestones is discussed, followed by lessons learned and a summary of the benefits of the new architecture. The concept of a shared Java codebase with platform specific endpoints is discussed, and current UMLet metrics are shown to prove the viability of the approach. A short general conclusion rounds off the discussion of the results.

9.1 Research Questions

The research questions have been formulated in section 1.2. The answers can be summarized as follows:

(R1) What software evolution approaches are feasible and how to evaluate their cost and risk criteria?

Software maintenance, aging, and evolution are relevant topics for every software project. Software ages, and it's important to invest time to limit the effects and keep control of the way the software evolves.

As Parnas [72] mentions, the failure to modify the program according to changing needs, as well as the result of changes can increase maintenance costs. Therefore it's important to think ahead and plan how the software should evolve. It's also important to invest time to keep software modules simple and maintainable by cleaning up convoluted code and by redesigning program logic when necessary.

Software aging has a strong relation to the topic of legacy code. According to Brodie, a legacy system is *"a system which significantly resists modification and evolution"* [14], which is very similar to the definition of an aged system. It's important to decide whether to preserve or redesign such a legacy system, and the answer to this question is not easy and depends on the actual system. Section 2.3 lists typical reasons to preserve or redesign a system.

The main reason why these topics are of such an importance, is that maintenance is costly and the maintainability of a system is linked to it's architecture and design. For example it's hard to maintain software without automatic tests, which means every change can potentially break features. Several different sources show how expensive maintenance can be: at least 40% of the initial development costs [15]; over 90% of the total costs of the system [76]; more than 70% of the software budget [42]. All of these sources have in common that the maintenance budget is substantial, which further underlines the importance of choosing the correct evolution approach for the system.

Section 2.5 shows different evolution approaches, which are extracted based on the previous findings. It's not possible to find a generic approach for all kinds of software projects, therefore the listing of software evolution criteria in section 2.6 shows positive and negative aspects for each of them.

Many of those factors are cross-cutting concerns, so it's important to consider feedback from all stakeholders when deciding the right way to evolve a system. It has to be evaluated by software architects, if the new architecture fits the system, the programmers must analyze new tools, customers and market analysts have to decide whether there is a possible market for the evolved software, and managers have to decide if the business environment enables the chosen approach.

Although the approaches and criteria are valid for any programming language, the language and it's ecosystem are important factors for the selection process. If tools and frameworks exist to facilitate the migration process, a tool-supported "soft" migration can be a viable approach.

In summary, it is important to analyze the technical background and the area of application of the software to be evolved. Decision drivers should be identified and the decision-making process should include all stakeholders, in order to extract several possible evolution approaches, of which one must be implemented.

(R2) How do these criteria affect an actual migration based on a large, real-world software package?

To provide an exemplary application of the previously defined software evolution approaches and criteria, the software project UMLet has been chosen. The evaluation of the decision drivers (section 4.1) suggests a migration to a more future proof platform than the Java virtual machine, but the application of the software evolution criteria to UMLet (section 4.2) shows that such a migration contains several risks.

Therefore a soft migration approach is chosen, by applying a Java to JavaScript transpiler (GWT), in order to reuse as much code as possible. This mitigates the migration risk of a declining user base, because the existing platforms stay active and maintained. It also avoids the maintenance overhead of duplicated code in different programming languages. The risk that browsers may not be able to run GUI heavy applications, such as UMLet, fortunately has been proven wrong, although modern browsers are required to run UMLetino.

After the migration, it can be concluded, that all of the milestones have been met and the chosen approach has been successfully implemented. The effects on stakeholders are: the software architecture has been redesigned, and the clear layering should make the application easier to understand for software architects and programmers. Programmers only have to acquire some GWT knowledge, if they plan to work on the web version, otherwise they must only comply with the restrictions on the shared codebase, which means unsupported portions of the Java API, are not allowed to be used there. Customers have a wider choice of platforms to use UMLet and it is no longer required to install Java to do so.

In terms of implementing the web version, there was more effort needed than initially estimated. This is mostly due to the different expectations users have when they use a standalone application, a web application on a desktop PC, or a web application on a mobile device.

As described in chapter 6, the design goals were clear, but it needed multiple iterations to make sure the web application felt natural. There are still several limitations of the web application in comparison to a native one, mostly in terms of performance and access to resources, such as the file system or the clipboard, and there is more work required to implement the missing features.

The general idea of separating the shared code from platform specific code worked out well in terms of increased maintainability. The code analysis shows that UMLetino shares most of it's code with the standalone application, which simplifies maintenance and reduces code redundancy.

9.2 Milestones

This section discusses the milestones from section 1.3 and checks if they have been met.

(M1) Project Modularization: Modularize the application by separating platform specific code from the shared code.

The implementation of M1 is described in chapter 5. The first important step of the modularization was the extraction of common functions for the elements. The basic idea is that a specific functionality, such as *color the background red* is a module (called Facet) which can be shared between several elements. Table 5.1 shows the available general purpose functions which are implemented as shared code.

The other important step is the abstraction of the drawing API to enable platform independent drawing calls. This drawing API is implemented once per platform. The implementation for UMLet Standalone and Eclipse Plugin redirects drawing calls to the Swing/AWT GUI framework, the implementation for UMLetino redirects drawing calls to an HTML canvas.

The separation into platform specific code and shared code has also paved the way to a straightforward implementation of a web version of UMLet.

(M2) Web Version Implementation: Design and implement a new web version of Umlet, while still maintaining the standalone and eclipse plugin versions.

As described in chapter 6, UMLet's core features have been successfully migrated to the web, but there is more work required to improve the usability and user experience of the web version, especially if mobile devices are the target audience.

In retrospective, the implementation of the web version was more elaborate than expected. This was mostly due to the limitations of browser applications (see chapter 3.3) and the requirement of keeping backward compatibility to old UMLet diagrams and elements, which makes it difficult to generalize several aspects of the application without huge time investments.

Another issue with the web version is that, compared with a standalone application, file operations are very limited and not typical for web applications. To make the web application feel more natural, UMLetino already provides Dropbox integration, but could be extended to other cloud storage solutions or an UMLet specific serverside which also provides a diagram storage platform (see chapter 10.3).

Finally, the support of mobile devices is more complicated than expected, because the interaction with UMLet is based on precise input, e.g., from a mouse cursor. Touch devices are too inaccurate for the current way of interaction and require their own solutions to be usable. There are also some technical problems with overlapping mouse and touch listeners and zooming (see chapter 10.3).

9.3 Lessons Learned

During the actual migration, the following noteworthy issues were encountered:

- **Front-end code is often more platform-dependent and should be de-coupled from business logic.** There are several graphical libraries for Java SE like AWT, Swing, or SWT. Android and GWT don't use those, but offer their own APIs. One possible way of avoiding this duplication is the usage of HTML (probably with some JavaScript generated by GWT), because most modern GUI frameworks can display embedded HTML+JavaScript views. In case of UMLet the code didn't have a clear separation between GUI and business logic; therefore a significant amount of time was necessary to modularize and decouple the components of the application in order to make the extraction of a shared core component possible. Fortunately, large portions of UMLet's graphical output is drawn on a Canvas where every platform offers its own implementation with only minor differences.

- **Usage of the Java API and 3rd-party libraries impact portability.** The Java API itself is extensive and not JVM based implementations often limit the range of supported classes (e.g., as mentioned before, AWT classes are not supported in GWT and Android). If 3rd-party libraries are used in shared code, they must follow to the same restrictions. In addition, new Java versions can add new APIs and syntax which may not work on all platforms. GWT imposes an addition restriction, because it compiles Java source code (and not byte code) to JavaScript, i.e., a 3rd-party library must be available as source code and not only as compiled classes.

- **Special language features like reflection and regular expressions limit portability.** GWT does not support reflection out of the box, and the default Java RegEx classes are only partially supported. Complex Regular Expressions must use GWT specific classes that work more like JavaScript RegEx than Java RegEx. In general, if a specific JVM feature like byte-code generation or just in time compilation is used, it has to be verified if it is supported by the target platform and the used transpiler.

- **GWT, while still evolving, is already suitable for complex web-based GUIs.** GWT is well documented and used for complex web GUIs of large companies such as Google (AdSense, Docs) or RedHat (Wildfly Management Console). Eclipse and IntelliJ IDEA plugins ease development and testing. The GWT Dev Mode makes debugging within the IDE very convenient, but is deprecated and restricted to older browsers (e.g., Firefox 26). Current browsers have removed support for the NPAPI (which is required for the dev mode), therefore developers should use the so called Super Dev Mode; an alternative which works without a browser plugin (because it compiles to real JavaScript and connects it to the Java code using Source Maps). The Eclipse integration is less good and requires a 3rd-party eclipse plugins (SDBG[1]), but instead it's possible to debug the real JavaScript code within the Chrome browser, which enables the developer to debug the real code and use the Chrome dev tools.

- **Useful web applications require modern browsers.** In general, web applications that should behave like standalone desktop applications typically require certain APIs to interact with the underlying system. This is a minor inconvenience for browsers like Chrome or Firefox, which get constantly updated, but other browsers like the Internet Explorer often lag behind. UMLetino also requires some specific HTML 5 features like the Web Storage API or the File Reader API, which are only available in Internet Explorer 10+.

- **Platforms have different constraints.** Although modern browsers offer several APIs to allow deep system integration, the web platform still has many constraints that do not exist for standalone applications. One example is the interaction with the file system. Standalone applications like UMLet have full access to the file system, but web applications have only limited access. File can be read by using the HTML 5 File Reader API, but most browsers disallow write access to the file system (only Chrome allows it to a sandboxed section of the filesystem). Other examples are Clipboard access, performance restrictions or hardware interaction with a microphone, camera, bluetooth devices, ...)

- **Mobile devices and touch interaction present a special challenge.** Although modern mobile browsers offer nearly the same feature-set as desktop browsers, the application needs special adaptations to be mobile friendly. There are many factors which make optimization for mobile devices hard like the screen size, touch interaction and zooming. UMLet's user interactions are based on precise targeting which is easy with a mouse but very hard with touch input. Double clicks and right clicks are used as specific actions for UMLet, but double tapping typically means zooming and right click is not possible. Therefore UMLet would need a completely different user interaction schema to be usable with touch inputs.

- **Evolution approach depends on software and its environment.** It's important to invest enough time to find the best evolution approach for a specific piece of software and its

[1] `https://github.com/sdbg/sdbg`

environment. For UMLet the Java core with several endpoints (including the new web based one) is working fine, but this is heavily influenced by UMLet-specific requirements, which made this decision the best one for UMLet (e.g., the programming language is Java, there exist a Java to JavaScript transpiler with a Java API emulation, UMLet's core logic is drawing on a canvas without using many GUI framework components, the browser performance and feature set is sufficient for UMLet, . . .)

9.4 Benefits of the New Architecture

There are several benefits for users and developers. Typically user benefits are also beneficial for developers, because consistent behavior comes from shared code which is more maintainable than copy & pasted code.

Benefits for Users (and Developers)

- Elements and diagrams work on all platforms
- Consistent functions on all platforms and elements (see table 5.1)
- Syntax completion for all functions
- More powerful customizations for elements (e.g. autoresize, wordwrap)

Benefits for Developers

- Project structure modularized with Maven
  - Code references only possible from top to bottom
  - Platforms (and tests) have separate dependencies
  - Tests and generated code no longer mixed with main codebase
  - Platform executables are simply compositions of modules
- Project no longer requires Eclipse IDE to build
- Eclipse P2 Update Site is generated during build
- JUnit tests executed during build. Build fails if test fails.
- Findbugs, Google ErrorProne executed during build
- Travis CI Continuous Integration Server constantly build project
- Removal of code duplication makes codebase easier to maintain

9.5 Java Based Core Component

The evaluation of the concept of having a Java based core component and separate platform specific endpoint implementations has shown the following strengths and weaknesses.

Strengths

- "Back-end code" (algorithms or business logic without UI interaction) is often easy to share, and therefore a good fit for this approach.
- Amount of duplicated code is reduced, therefore maintainability and consistency of program behavior on all platforms is improved (DRY Principle: *"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."* [45]).
- Java API and its ecosystem (IDEs, libraries, annotation processors, ...) can be accessed, even when developing a web application (JavaScript doesn't provide such an extensive standard API and tools).
- All (or most) code is written in Java, which is a good pick for shared code, because it runs on platforms with a JVM or Android VM (Dalvik or ART) and can be transpiled to JavaScript using GWT, and to Objective-C using J2ObjC.

Weaknesses

- "Front-end code" often cannot be shared (e.g., Swing text input component differs from HTML text input component).
- Shared codebase is limited to the subset of the Java API which work on all target platforms (e.g., GWT [29] and Android [71]). Missing classes must be replaced with self written versions or wrappers to be shared (see chapter 5.4).
- Behavior and Performance of Java APIs may deviate between platforms (bugs may only exist on some platforms, behavior of complex APIs may differ - e.g., Java RegEx support is limited in GWT).
- Build process may become complicated due to the transpiler and the multi-platform nature.

Fortunately the weaknesses are not very limiting for many types of software. In case of GWT and Android, the most important Java API is supported and the missing classes have been recreated without much effort in the shared codebase.

The chance of critical bugs on one platform is not very high, because both GWT and Android are widely supported and used in the industry, therefore such a bug would be detected and fixed soon.

Typical Use Cases

- Legacy Java desktop applications, which should be migrated to the web (because other solutions would require a full rewrite because of the language gap)
- Multi-platform Java applications with dynamic UI or little platform specific UI (e.g., drawing tools, games, pdf viewer, physics simulation, ...)

The first use case describes a potential evolution scenario for tools such as UMLet. The second one shows that this concept can also be used to develop new cross-platform applications. Ray Cromwell gave a presentation at the GWT.create conference in January 2015 [24], where he explains how Google approached the development of their product Inbox. He mentions that all Inbox applications (Android, iOS and web) share 60-70% of their code using a Java-based core component.

9.6 Current Metrics of Code Reuse

Another indicator, that the idea of a Java based core component is working, is shown in figure 9.1. While chapter 7.3 discusses the statistics immediately after the migration, these numbers include subsequent changes to the codebase and present a current snapshot:

- UMLetino consists of 87% shared code (so only 13% are platform specific)
- Standalone UMLet and the Eclipse Plugin share around 50% of the code with UMLetino; both share most of the code with each other.
- The Eclipse plugin integration has been improved, but the additional code is only affecting the eclipse plugin module.
- The elements module has grown the most, because another contributor has redesigned the sequence all in one diagram and added a lightweight custom element variant (every element is now customizable via drawing functions).

In summary the figure shows, that most new code can be implemented in a platform independent way. Platform specific additions (such as the improvements to the Eclipse plugin) are also more maintainable, because they are clearly separated from the rest of the code.

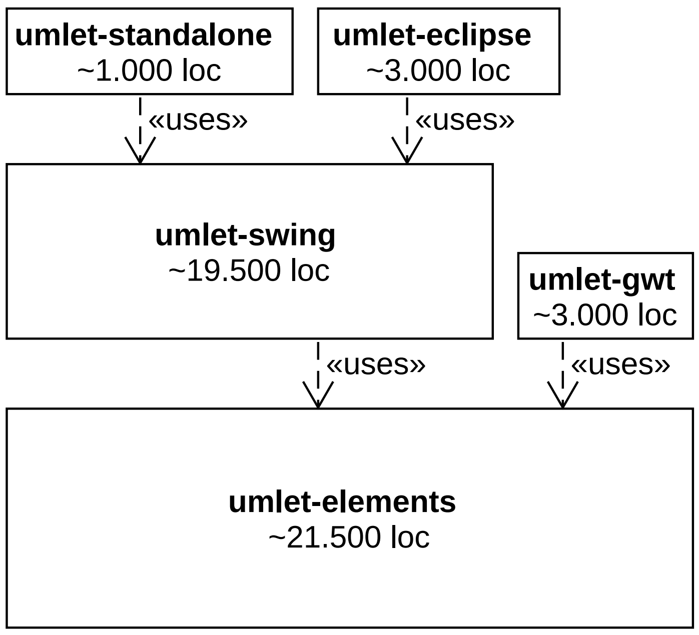


Figure 9.1: Lines of Code per Maven Module (2016-09-18)

9.7 Conclusion

Software maintenance, aging, and evolution are integral parts of the modern software lifecycle.

This thesis discusses several possible software evolution approaches and provides a list of criteria to support stakeholders in finding the best way to evolve a specific piece of software.

Ideally, these considerations should not be an afterthought, but instead should be made during the software's initial design stages, to make sure the chosen path of evolution is well though out and appropriate for the software and its environment.

The decision to evolve UMLet by extending the available platforms to the web, while still keeping the existing ones intact, has been proven successful. Time will show if the new web platform only extends the user range by introducing UMLetino to tablets, chrome-books and other devices, or if it will replace standalone UMLet at some point in the future. In any case the redesign of the code base has improved its maintainability and modularity, and therefore created a sound basis for the future of UMLet.

CHAPTER 10

Future Work

UMLet has improved a lot due to the reengineering efforts and the migration to the new platform, but there is still potential for future enhancements. There are general improvements, such as the usage of the dependency injection, in order to make the code easier to understand and test, but there are also platform specific improvements listed.

UMLet Standalone and the Eclipse plugin would benefit from a refactoring of the Swing Listener structure and the evaluation of alternatives to Swing (e.g., JavaFX).

UMLetino would benefit from the introduction of a change history, an auto save feature, more export formats and usability improvements, especially for mobile devices with touch interaction. Finally other web based UML tools can be used as inspiration for further enhancements of UMLetino.

10.1 General Improvements

As mentioned in chapter 2.7, there are several useful tools which can improve the code quality. The following sections suggest some applications for these tools:

Dependency Injection

During the work on UMLetino, many parts of the GridElement code have been changed and manual dependency injection has been introduced to make them reusable between Swing and GWT (e.g., by injecting the DrawHandler and other platform specific classes), but other classes are still tightly coupled to their dependencies and therefore less testable and reusable as possible.

As manual Dependency Injection can be tedious in larger projects (e.g., Singletons which are currently fetched using **getInstance()** which must then be passed through many classes in the dependency graph), Dagger 2[1] could be used as a simple dependency injection framework which works in Java and GWT.

[1]`https://google.github.io/dagger/`

If a dependency injection framework is used, each platform (Eclipse, Swing, GWT) will have one separate configuration which binds the required platform-specific-implementations for interfaces in the shared codebase. In addition it's much easier to write Unit Tests for parts of the program, because the dependency can be simply overwritten with a test-implementation or a mock of the class.

Automated Tests

Currently UMLet has only a very small amount of Unit Tests. The main reason is that the tight coupling of the components make it hard to test them separately.

As soon as more components are better decoupled using Dependency Injection, it is possible to write more tests to ensure components keep working as expected.

Migration of the Activity All In One Diagram

While the Sequence all in one diagram has been ported to the new architecture by another student, the Activity all in one diagram is not yet GWT compatible and strongly tied to Java Swing/AWT classes. As soon as it has been ported, all of the old diagrams are available in the new architecture.

Mouse Cursor

Currently the mouse cursor handling is completely separated. This is mostly due to the fact that they use different constants for each GUI framework. The problem with this approach is that an element cannot change the cursor if the mouse is over a specific part of an element and that the mouse cursor change logic must be duplicated in each endpoint project.

To solve these issues, an enumeration can be created which is used in in the shared code and mapped to the concrete GUI framework using a converter. The same approach has been used in chapter 5.4 for the creation of missing basic classes like a **Color** class.

Commands (Add and Remove Element, Copy, Paste, . . .)

As described in figure 4.1, commands are undo-able interactions with the diagram like Add Element, Remove Element, Copy, Paste, . . . which are based on the Command pattern, which is described in [39].

Currently they are located in the endpoint projects, because the Change History is currently not implemented in UMLetino (see chapter 10.3) and because they contain implementation details of some OldGridElements. To avoid potential incompatibility issues, they should only be moved to the shared code after the OldGridElements have been successfully migrated and removed from the codebase (see chapter 10.2).

Even if most of the command related code is moved to the shared codebase, some parts will remain specific for the endpoint implementation like the Change history (see chapter 10.3), therefore some commands will only share an interface or abstract super class but still need a concrete subclass for each endpoint implementation.

10.2 UMLet Standalone and Eclipse Plugin

As the main focus of this thesis was the creation of UMLet as a web application while retaining the functionality of the standalone program with its legacy elements, there are some potential improvements to the Swing specific code base and to overall concepts of the program.

Swing Listeners

There are two main problems with the Swing listener approach:

1. A subclass of the listener hierarchy often overwrite behavior of the super class which makes the behavior of the listeners hard to understand.
2. The Swing listener logic which decides the event recipient for overlapping elements is not compatible with UMLet's logic (elements on a higher layer have priority and in case of the same layer, smaller elements have priority over larger ones).

To solve the second problem with the current Swing listener approach, the Swing specific part of the GridElements must overwrite the javax.swing.JComponent.contains() method.

UMLetino uses a much simpler approach with only one generic diagram listener. If the mouse is clicked, it is calculated which GridElement is the recipient of the event (based on the mouse position and the layer and size of the elements on that position) and the platform independent event-handling method of the GridElement is called.

This approach could also work for the Swing listeners and would make it possible to move the bulk of the event-handling code into the shared codebase.

It would also make the implementation of a dedicated zoom layer much easier, because the listeners are the only reason why each GridElement must be a separate javax.swing.JComponent. This restriction limits the flexibility of such a zoom layer which could otherwise be completely placed in the shared codebase.

Removal of OldGridElements

Currently all of the OldGridElements still work if they are part of an old diagram, but only the CustomElement and the Activity all in one diagram element are not deprecated (because those are currently not replaced with NewGridElements).

Deprecated elements (such as the old UseCase or the old Relation) have been removed from the palettes and a deprecation message is shown if such an element is selected.

It should be evaluated how and when these deprecated elements should be removed from the code base and what should happen if a diagram with such an element is opened.

Two possible approaches are:

- simply remove them from the codebase in a future version and show an error instead of the element if an old diagram is opened.
- migrate every deprecated element to a matching new one. This could either be part of UMLet or part of a separate migration-batch program which transforms elements in uxf

files. This approach is more complex than it sounds, because the syntax of some functions have been unified between elements and therefore have been fundamentally changed.

Evaluate Alternatives to Swing

Since Java 7 Update 6 [65], Java FX is contained in the JDK and JRE, and according to the official JavaFX FAQ [75], it is replacing Swing as the new client UI library for Java SE.

Although Swing will still be part of Java SE for the foreseeable future [75], it's time to look into alternatives.

As the official successor, JavaFX is the obvious choice for building a modern Java GUI, but there are other alternatives like a plain SWT or the SWT based Eclipse 4 RCP GUI could be looked into.

It is possible to combine JavaFX and Swing [75], as well as SWT and Swing (which is already used for the UMLet Eclipse Plugin), therefore a gradual migration of the Swing specific codebase could be done.

10.3 UMLetino (Web) Future

This section will focus on suggested changes to improve the user experience and feature-set of the web version of UMLet.

Change History (Undo and Redo)

Contrary to the standalone version, the web version currently doesn't support change history.

The code which handles the change history is currently strongly coupled to other parts of Swing specific code and therefore cannot be simply moved to the shared code base.

One approach to solve this issue is to define the shared code (e.g., an interface or abstract class) within the shared project and the implementation specific part in the endpoint project.

There are several ways to implement a change history in the web version, as described subsequently.

URL Based History

One feature specific to web applications is the URL. In UMLet we don't really have resources which need locations, but we can use the URL to represent the historic state of the diagram and therefore embed our change history in the browsers page history. A simple type of representation would be a number which would look like:

```
<base_url>?history=5
```

An undo action would go one step back to

```
<base_url>?history=4
```

A redo action would go one step forward to

```
<base_url>?history=6
```

This simple concept would allow for a browser supported history handling and therefore would feel natural to the user of the application.

It would also allow the user to use additional browser history features like listing the history of visited URLs by holding the mouse button down on the back-button in the browser. This would allow the user to undo multiple steps at once.

Of course the typical shortcuts for history handling (CTRL+Z and CTRL+Y) should also be supported and they would trigger the same action as a browser-back and browser-forward action.

Auto-Saving as Persistent History

If the change history is only stored in memory, a computer or program crash means that the history and the current state of work is lost.

A simple solution to this issue would be a scheduled automatic save of the current diagram. The saving could happen time-based (e.g., every 5 minutes), diagram-action-based (e.g., every 20 actions like move, resize, property change, . . .) or user-action-based (e.g., autosave if the user closes the tab). There must be a maximum amount x of automatically saved diagrams (e.g., 10) and if the x+1. autosave happens, the oldest one would be removed. This could be implemented in addition to the change history and would provide simple persistent “stored points” in the work history of the user.

One potential issue of this approach is the size limit of the browsers local storage. According to the W3C Web Storage recommendation [78], a limit of five megabytes per origin is recommended, but this suggestion can be updated in the future. As the textual data of Strings is considered to be UTF-16 [90], we can assume that some browsers will also store the strings in UTF-16.

This is important, because UTF-16 uses at least 2 bytes per character, while UTF-8 can represent ASCII characters with only 1 byte [30]. As ASCII characters are the most common characters in typical UXF diagram files (which is also the format used to store diagrams in local storage), that means a diagram stored in local storage (possibly in UTF-16) needs approximately 2x the space it would need in a file on the hard drive (stored in UTF-8).

A practical test on current browsers (Firefox 32, Chrome 37, Internet Explorer 11) using [2] has shown that all browsers provide roughly five megabytes of local storage. For a simple estimation of the maximum amount of storable diagrams in typical browsers, we assume that usually uxf diagrams use 10-50kb if saved as UTF-8 files. which will mean they use 20-100kb in local storage while having 5mb available.

Therefore in average we should at least be able to store 50 large diagrams or 250 small ones in local storage, which should be enough for most users even if 10 of them are “reserved“ for automatically saved diagrams.

[2]Website to check the storage limit for different types of Web Storage: `http://dev-test.nemikor.com/web-storage/support-test/`

More Export Formats

Currently the export is limited to UXF and PNG (which is supported by default in an HTML Canvas). To catch up with the multitude of formats UMLet can export, the most important alternatives should be investigated:

- **SVG:** There is the javascript library **canvas2svg** [3]. It cannot directly export an HTML canvas to SVG (this would not look very good because an HTML canvas is bitmap and not vector based), but instead offer an alternative element with most of the canvas-functions. One way to use it in UMLet would be to implement a **GwtSvgDrawHandler** variant which draws into this canvas-like element instead of a real HTML canvas.
- **SVG Alternatives:** A library specific for GWT is **lib-gwt-svg** [4]. Alternatively, there is also **Raphaël** [5] and some libraries to use it with GWT [6] and the probably discontinued library **gwt-graphics**[7].
- **PDF:** There is the javascript library **jsPDF** [8] which exports an HTML Canvas into a PDF. Unfortunately this would only export the bitmap based HTML canvas, therefore it wouldn't look as good as the vector based PDF export of UMLet, but perhaps it could be applied to a SVG output.

Performance Improvements

Although some performance improvements have already been implemented in UMLetino (see chapter 6.11), there are further options which should be explored.

Viewport Culling

Typically performance is very good for smaller diagrams and decreases slightly with every added diagram element, because there is more stuff to calculate and to draw.

At some point the required space of a diagram will grow larger than the space which is viewable in the browser window. The visible part of the whole diagram is called the **Viewport**.

The term **Viewport Culling** means that because the user can only see the visible part of the diagram, we can cull all not visible parts of the diagram, which means in the case of UMLet that we don't have to draw non visible diagram elements on the canvas.

A simple way to implement **Viewport Culling** would be to draw only elements which have a not empty intersection set with the viewport, although this simple approach would only cull elements which are completely outside of the viewport and still draw elements which are partially outside as a whole.

[3]canvas2svg: `https://gliffy.github.io/canvas2svg/`

[4]lib-gwt-svg: `https://github.com/laaglu/lib-gwt-svg`, samples: `https://github.com/laaglu/lib-gwt-svg-samples`

[5]Raphaël: `http://raphaeljs.com/`

[6]RaphaelGwt `https://code.google.com/p/raphaelgwt/` or Raphael4Gwt `http://code.google.com/p/raphael4gwt/`

[7]gwt-graphics: `https://code.google.com/p/gwt-graphics/`

[8]jsPDF: `https://github.com/MrRio/jsPDF`

Partial Canvas Clearing

At the moment the whole canvas is cleared and all diagram elements are drawn on it after every change on the canvas. Even with **Viewport Culling** the whole visual area would still be redrawn every time.

A possible, albeit more complex performance improvement would be clearing only the part of the canvas which has changed. The problem with this approach is that cascading changes must also be considered.

E.g., Element A partially overlaps with Element B. Even if only Element A changes, both elements must be cleared and redrawn, because otherwise a part of Element B would not be drawn.

Another example for cascading changes are sticking relations. If an element with a sticking relation is moved around the diagram, the relation will change, although it's completely outside of the element space.

Lastly, a HTML canvas is typically cleared when resized, which happens quite often in UMLet (e.g., if an element is moved out of the current diagram border and therefore resizes the canvas)

As a result, this technique would be much harder to implement than **Viewport Culling**, while the additional performance gain is limited by the visible part of the diagram.

Usability Improvements

Clipboard

Although there is a W3C Working Draft on the Clipboard API[9], only Firefox currently supports it according to [10].

If the Clipboard API is supported by many browsers, it may make sense to use it instead of the local storage based solution which is currently used (see chapter 6.8)

Configuration and User Customization

In Standalone UMLet, there is a configuration file which is basically a key-value map to customize the program for the user. In web applications, there is no general filesystem access and therefore no configuration files.

Although there are several alternative approaches which work well for web applications:

- **URL Parameters**: a simple approach which can easily be shared between different computers and users. Disadvantage: The user must manage the configuration with his bookmarks.

- **Local Storage**: the local storage is already used extensively, therefore it would be simple to store the configuration there. Disadvantage: Not shareable between computers and users.

[9]W3C Working Draft on Clipboard API: `http://www.w3.org/TR/clipboard-apis/`

[10]caniuse Clipboard API: `http://caniuse.com/clipboard`

- **Server side storage**: A server-side component could offer a user login with several data like the configuration. Disadvantages: currently there is no server-side component of UMLet, also it would not be possible to get the configuration if the user is offline.

Mobile Devices

AsUMLetino is pure JavaScript, it runs on any modern browser which supports the required HTML 5 features. It has been tested and runs smoothly on a **Samsung Galaxy Nexus** using **CyanogenMod v11** using one of the following two browsers: **Android Browser v4.4.4** or **Android Chrome v37.0.2062.117**.

If UMLetino is run on a tablet with attached keyboard and mouse, it should deliver the same user experience as on a desktop computer, but if used on a small screen device with touch input, there currently are some problems with touch listeners and zoom handling of the mobile OS.

The following list contains the main problems which I have experienced (testing on an Android device) while developing the currently limited support for mobile devices:

- **UMLet interaction is based on precise targeting** which is easy with a mouse, but much harder with touch input (especially on small screens). The main problems are too small menu items, resize-hooks on the elements border and especially relation handling (like dragging relation ends around, dragging out new relation points, . . .).

- **Double-tap is default for zooming**. With a mouse, single click and double-click are typical interactions which are also used for core-UMLet features like “copy element“. On a touch-device, a double-tap is reserved for zooming.

- **Right click not available**. On a touch device, there must be an alternative to show the context menu, which can be, e.g., “touch for x ms without moving“.

- **“Touch and move finger“ moves web app around if zoomed in**. This is a problem because UMLet extensively uses “click, hold button and move“ actions which translate to “touch and move finger“, therefore it’s hard to decide when such an action should be resolved by UMLet and when it’s only for moving around the zoomed-part of the application.

- **Mouse listeners interfere with touch listeners**. There are some problems, if the web application listens to both mouse and touch listeners (e.g., a mouse-move-event is triggered on each touch-down-event). The currently implemented workaround is to remove all mouse listeners as soon as a touch is registered.

- **No diagram zooming**: Currently UMLetino uses the built in zooming support of browsers. While this works on a desktop computer, mobile browsers typically don’t offer zoom support.

These problems show that it’s not an easy task to improve UMLet’s user experience on mobile devices, but a possible way could be a separate CSS file and several adaptions (like larger “interaction areas“ for elements) if a touch device is found (or after the first touch has been registered).

As mobile platforms are challenging for many GWT developers, there are some tools which ease development for mobile devices like **mgwt** which provides mobile widgets, touch support and more, as well as **gwt-phonegap** which allows deploying GWT apps as native apps in an app store. More information about these tools can be found on their website `http://www.m-gwt.com/` and in a presentation from Nicholas Chen [21].

Zoom

Although the browser zoom works fine for the diagram, there is the issue that all elements of the web page are zoomed. Although it still looks good in case of the palette, the menu items and the properties panel are too small if zoomed out.

As there is also the issue of missing browser zoom support on mobile devices, perhaps there should be a specific zoom implementation in UMLetino which does not rely on the browser anymore and only zooms the diagram and/or palette.

Server Side

Currently UMLetino is a pure JavaScript application which will also work in offline-mode. Although this concept has its advantages, a server-side component would make many interesting features possible.

Potential Features of a Server Side

- **Diagram Storage**: If diagrams can be saved on a server, they are much easier to share between users. A typical usecase can be embedding the diagram URL into a projects documentation (advantage over image embedding: **Single Source of Truth**; the diagram can be changed without updating the documentation).
- **Configuration Storage**: the configuration can be shared easily between different computers (by sharing the URL).
- **Custom Palettes**: Currently the palettes are fixed and contained in the JavaScript code. A server-side component would allow to create and use custom palettes (as in Standalone UMLet).
- **Diagram Export**: Diagram export from HTML Canvas is limited to fewer formats than Standalone UMLet and it’s not possible to trigger a file-download-pop-up for the browser. Such features can be implemented by sending the diagram to the server and returning the requested export format.
- **Custom Elements**: Custom Elements consist of Java code which is compiled at runtime which is not possible in JavaScript. This can also be fixed by sending the element to the server, compile it and send the result back.

Inspirations from other Web Based UML Tools

As described in section 3.7, there exist other web based UML tools with different concepts of user interaction and diagram creation. Therefore it is useful to analyze their way of doing things and see if some of the concepts make sense for UMLet.

Syntax

The syntax of **yUML** uses to create Class and Use Case diagrams is very simple and intuitive and would also make sense for UMLet which currently does not support those 2 diagrams in all-in-one-style. In addition the Activity diagram syntax from **yUML** and the Sequence diagram syntax from **js-sequence-diagrams** and **websequencediagrams** can be compared to the UMLet syntax for these diagrams and simplifications should be applied where they make sense.

For example, UMLet requires the user to specify an ID for each process and then work with the ID, but **js-sequence-diagrams** and **websequencediagrams** use the name of the processes instead. Although the ID concept is more stable in terms of refactoring (if you change the name, you only have to change it at one place, because the ID can stay the same), the name-based system is more intuitive and easier to write, especially for simple diagrams.

User Interaction

Another interesting aspect is the user interaction with diagram elements.

Aside from the tools which are completely text-based with no interaction at all, there are some interesting concepts, especially regarding touch interaction.

sketchboard's way of handling relations is very intuitive and touch-device friendly as you can simply drag them out of one element and if the target element doesn't exist, it will suggest a list of elements which can be created. UMLet's relations instead require much more mouse interactions, because every point on the relation line must be placed manually. Also the interactions must be very precise which is challenging on touch devices (especially on smartphones with small displays).

Text Based Diagram Representation

The idea of creating simple text based representations of diagrams (like **asciiflow** does) is very interesting, because it enables storing the diagram in a text based format which is much more human-readable than the current XML based format.

In addition to that it is very simple to embed the saved diagram in any kind of text-file based document.

Even though this kind of diagram storage solution is only usable for small sized diagrams (otherwise the file-size would grow large very quickly due to many spaces) and simple diagram types (displaying a class diagram in ASCII is much simpler than displaying a state diagram) it is definitely an interesting feature which should be kept in mind.

APPENDIX A

UMLet Terminology

DrawHandler

The DrawHandler is an abstract unified drawing API, which is independent from the underlying graphical framework (Swing, HTML Canvas, . . .)

This abstraction layer is necessary to move the drawing code for the GridElements into the shared codebase, while ensuring, that the elements look the same on all platforms. More details can be found in section 5.3.

Facet

A Facet consists of a method which checks if the current line should trigger the facet action and the action itself.

E.g., the line **fg=red** triggers the ForegroundColorFacet (because it triggers if a line starts with **fg=**) and sets the foreground color of the element to the given value. More details on Facets can be found in section 5.1.

GridElement

GridElement is the generic term for any kind of element which is drawn on the diagram (e.g., a UML Class, UML Use Case, Plot, . . .). The name represents the generic nature of the idea that there is a grid on the diagram and any kind of element can be drawn on it. More details about the elements can be found in section 3.6.

In the Java code, GridElement was an abstract super class for all elements and extended JComponent itself. With the separation of the projects and the removal of the Swing dependency, it has been changed to an interface which is now shared by old and new GridElements. This has the advantage that code which interacts with GridElements doesn't need to know if its an old or new one.

OldGridElement

The old elements are now called OldGridElement and still extend JComponent and are therefore still tightly coupled with the Swing GUI library.

Besides the tight coupling to Swing, they also contain large portions of copy & paste code, therefore it was not possible to simply change the behavior of functions (such as fg=red) without changing it in the code of every single OldGridElement.

They will be removed at some point in the future (see chapter 10.2).

NewGridElement

The new elements are independent from any kind of GUI library as the draw specific code has been moved to the DrawHandler interface and the GUI-component-specific code is moved to a Component interface which are both implemented separately for each GUI library.

Plotlet

Plotlet was a spin-off project, which was created as part of the bachelor thesis [37] together with other students. It used large portions of UMLet's codebase, as well as the general idea of transforming text into a graphical representation.

Since UMLet v12.2, Plotlet's functionality has been merged with UMLet and is now represented as its own palette. The plot elements have also been migrated to NewGridElements and therefore are also available in the web based UMLet version.

UMLet

This is the name of the Java Swing standalone application and the UMLet Eclipse plugin. The official name of the new web version is UMLetino, but all projects share one GitHub repository and issue tracker, therefore UMLet may sometimes be used an umbrella term for all variants of UMLet, including the web version.

UMLetino

This is the name of the GWT based web version of UMLet. The most current version can be accessed at `http://www.umletino.com`. All of the uxf diagrams which are only using NewGridElements can be opened using UMLet and UMLetino.

UXF (file format)

UXF is an XML based format which is used by UMLet and UMLetino to store diagrams in files. UXF files are interchangeable between these 2 programs as long as they only contain NewGridElements.

Bibliography

[1] The RedMonk Programming Language Rankings: June 2015. `http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/`. Accessed: 2015-12-03.

[2] Jay Lyman a senior analyst for enterprise software at 451 Research. `http://www.linux.com/news/enterprise/cloud-computing/731454-docker-a-shipping-container-for-linux-code`. Accessed: 2016-05-11.

[3] Web Browser Distribution according to NetMarketshare. `http://www.netmarketshare.com/browser-market-share.aspx?qprid=2&qpcustomd=0`. Accessed: 2016-07-10.

[4] Martin Auer and Stefan Biffl. A Software Modeling Capability Proxy Metric. In *Proceedings of the 5th International Conference on Research Challenges in Information Science*, pages 1–6, 2011.

[5] 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*, pages 466–473, 2007.

[6] Martin Auer, Johannes Pölz, and Stefan Biffl. End-User Development in a Graphical User Interface Setting. In *Proceedings of the 11th International Conference on Enterprise Information Systems*, pages 5–14, Berlin, 2009.

[7] Martin Auer, Thomas Tschurtschenthaler, and Stefan Biffl. A Flyweight UML Modelling Tool for Software Development in Heterogeneous Environments. In *Proceedings of the 29th Conference on EUROMICRO*, pages 267–272, Washington, DC, USA, 2003. IEEE Computer Society.

[8] B S Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE ’95, pages 86–95, Washington, DC, USA, 1995. IEEE Computer Society.

[9] Plugin based content doesn’t work on Chrome. `http://support.google.com/chrome/answer/6213033`. Accessed: 2016-05-11.

[10] Victor Basili, Lionel Briand, Steven Condon, Yong-Mi Kim, Walcélio L Melo, and Jon D Valett. Understanding and Predicting the Process of Software Maintenance Releases. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 464–474, Washington, DC, USA, 1996. IEEE Computer Society.

[11] Keith H. Bennett and Václav T. Rajlich. Software Maintenance and Evolution: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[12] Omar Benomar, Hani Abdeen, Houari Sahraoui, Pierre Poulin, and Mohamed Aymen Saied. Detection of Software Evolution Phases Based on Development Activities. In *Proc. 23rd IEEE Int. Conf. on Program Comprehension (ICPC)*, 2015.

[13] Robert V Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[14] Michael L. Brodie and Michael Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann, San Francisco, 1995.

[15] Frederick P. Brooks Jr. *The Mythical Man-Month*. Addison-Wesley, Boston, 1995.

[16] David Budgen. *Software Design*. Addison-Wesley, 2 edition, 2003.

[17] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, Chichester, UK, 1996.

[18] John Businge, Alexander Serebrenik, Mark Van Den Brand, and Mark van den Brand. An Empirical Study of the Evolution of Eclipse Third-party Plug-ins. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, volume 2009 of *IWPSE-EVOL '10*, pages 63–72, New York, NY, USA, 2010. ACM.

[19] Theodore Chaikalis and Alexander Chatzigeorgiou. Forecasting Java Software Evolution Trends Employing Network Models. *IEEE Transactions on Software Engineering*, 41(6):582–602, 2015.

[20] Nicholas Chen. Convention over configuration `http://softwareengineering.vazexqi.com/files/pattern.html`. Accessed: 2016-05-11.

[21] Nicholas Chen. mgwt & gwt-phonegap - gwt's future on mobile `http://www.daniel-kurka.de/talks/gwtcreate/mgwt.pdf`. Accessed: 2016-05-11.

[22] Elliot J. Chikofsky and James H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[23] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using Metrics to Evaluate Software System Maintainability. *IEEE Computer*, 27(8):44–49, 1994.

[24] Ray Cromwell. Google inbox: Multi platform native apps with gwt and j2objc `https://drive.google.com/file/d/0B3ktS-w9vr8IS2ZwQkw3WVRVeXc`. Accessed: 2016-05-11.

[25] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann, San Francisco, 2002.

[26] GWT Documentation. `http://www.gwtproject.org`. Accessed: 2016-05-11.

[27] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships. *2007 IEEE International Conference on Software Maintenance*, pages 94–103, oct 2007.

[28] W S El-Kassas, B A Abdullah, A H Yousef, and A M Wahba. Enhanced Code Conversion Approach for the Integrated Cross-Platform Mobile Development (ICPMD). *IEEE Transactions on Software Engineering*, 42(11):1036–1053, nov 2016.

[29] GWT Java Runtime Library emulation. `http://www.gwtproject.org/doc/latest/RefJreEmulation.html`. Accessed: 2016-05-11.

[30] The Unicode Consortium UTF FAQ. `http://www.unicode.org/faq/utf_bom.html`. Accessed: 2016-05-11.

[31] Eclipse Marketplace Top Favorites. `http://marketplace.eclipse.org/favorites/top`. Accessed: 2016-05-11.

[32] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, New Jersey, 2004.

[33] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. *IBM Research Report*, 25482, 2014.

[34] Martin Fowler. `http://www.martinfowler.com/articles/injection.html`. Accessed: 2016-06-04.

[35] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.

[36] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999.

[37] Andreas Fürnweger. Textbasierte Steuerung komplexer GUI-Interfaces - Architektur, Design und Entwicklung des Plot-Tools Plotlet auf Basis des Modellierungs-Tools UMLet. Bachelor's thesis, Vienna University of Technology (TU Vienna), 2011.

[38] Andreas Fürnweger, Martin Auer, and Stefan Biffl. Software Evolution of Legacy Systems - A Case Study of Soft-migration. In *Proceedings of the 18th International Conference on Enterprise Information Systems*, pages 413–424, 2016.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA, 1994.

[40] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing Energy Code Smells with Reengineering Services. In *Beitragsband der 42. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*, volume 208 of *LNI*, pages 441–455. GI, 2012.

[41] Object Management Group. `http://www.omg.org/`. Accessed: 2016-05-11.

[42] Warren Harrison and Curtis Cook. Insights on Improving the Maintenance Process Through Software Measurement. In *Proceedings of the International Conference on Software Maintenance*, pages 37–45, nov 1990.

[43] Israel Herraiz, Daniel Rodriguez, Gregorio Robles, and Jesus M. Gonzalez-Barahona. The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review. *ACM Computing Surveys*, 46(2):1–28, 2013.

[44] Matthias Hirzel and Herbert Klaeren. Code Coverage for Any Kind of Test in Any Kind of Transcompiled Cross-platform Applications. In *Proceedings of the 2Nd International Workshop on User Interface Test Automation*, INTUITEST 2016, pages 1–10, New York, NY, USA, 2016. ACM.

[45] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Boston, 1999.

[46] Shortcuts in Umlet Github Wiki. `https://github.com/umlet/umlet/wiki/Shortcuts`. Accessed: 2016-05-11.

[47] International Standards Organisation (ISO). *ISO/IEC 14764:2006 - Software Engineering – Software Life Cycle Processes – Maintenance*. ISO/IEC, 2006.

[48] Web Hypertext Application Technology Working Group (WHATWG) Web Workers Introduction. `https://html.spec.whatwg.org/multipage/workers.html#introduction-15`. Accessed: 2016-05-11.

[49] Saad Mousliki Use JavaScript, WebRTC API to access your Camera, and Microphone from browser. `http://www.codeproject.com/Articles/486354/Use-JavaScript-and-WebRTC-API-to-access-your-Camer`. Accessed: 2016-05-11.

[50] Kalpana Johari and Arvinder Kaur. Effect of Software Evolution on Software Metrics: An Open Source Case Study. *ACM SIGSOFT Software Engineering Notes*, 36(5):1–8, 2011.

[51] Brian W. Kernighan. *The C Programming Language*. Prentice Hall, New Jersey, 1988.

[52] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 151–160, New York, NY, USA, 2011. ACM.

[53] Stephan Kleuker. *Grundkurs Software-Engineering mit UML*. Vieweg, 2008.

[54] R Lämmel. Towards Generic Refactoring. *RULE '02 Proceedings of the 2002 ACM SIGPLAN workshop on Rule-based programming*, 2002.

[55] Jannik Laval and Stéphane Ducasse. Resolving Cyclic Dependencies between Packages with Enriched Dependency Structural Matrix. *Software: Practice and Experience*, pages 1–24, 2012.

[56] Meir M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. In *Proceedings of the IEEE*, volume 68, pages 1060–1076, sep 1980.

[57] Meir M. Lehman, Juan F. Ramil, Paul D. Wernick, Dewayne E. Perry, and Władysław M. Turski. Metrics and Laws of Software Evolution - The Nineties View. In *Proceedings of the 4th International Symposium on Software Metrics*, METRICS '97, pages 20–30, Washington, DC, USA, 1997. IEEE Computer Society.

[58] Meir M. Lehman, Juan Fernandez Ramil, and Goel Kahen. Evolution as a Noun and Evolution as a Verb. In *Proceedings of the Workshop on Software and Organisation Coevolution (SOCE)*, 2000.

[59] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley, Boston, 1980.

[60] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[61] Tom Mens and Serge Demeyer. *Software Evolution*. Springer, Berlin, 2008.

[62] merriam-webster.com Definition of Legacy Code. `http://www.merriam-webster.com/dictionary/legacy`. Accessed: 2015-07-05.

[63] Ludwig Meyer. Lightweight UML Model Creation. Master's thesis, Vienna University of Technology (TU Vienna), 2011.

[64] Akito Monden, Shin-ichi Sato, Ken-ichi Matsumoto, and Katsuro Inoue. Modeling and Analysis of Software Aging Process. In *Product Focused Software Process Improvement SE - 15*, volume 1840 of *Lecture Notes in Computer Science*, pages 140–153. Springer, Berlin, 2000.

[65] Java 7 Update 6 Release Notes. `http://www.oracle.com/technetwork/java/javase/7u6-relnotes-1729681.html`. Accessed: 2016-04-01.

[66] List of Browsersupport for FileWriter API. `http://caniuse.com/#search=filewriter`. Accessed: 2016-05-11.

[67] List of Java Virtual Machines. `http://java-virtual-machine.net/other.html`. Accessed: 2016-05-11.

[68] PYPL PopularitY of Programming Language. `https://pypl.github.io/PYPL.html`. Accessed: 2015-12-03.

[69] List of UML diagrams. `http://www.uml-diagrams.org/uml-24-diagrams.html`. Accessed: 2016-05-11.

[70] Oracle Java Card Technology Overview. `http://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html`. Accessed: 2016-04-01.

[71] Android Java API Packages. `https://developer.android.com/reference/packages.html`. Accessed: 2016-05-11.

[72] David Lorge Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Sorrento, Italy, 1994. IEEE Computer Society Press.

[73] Johannes Pölz. UML Diagram and Element Generation - Exemplary Study on UMLet. Master's thesis, Vienna University of Technology (TU Vienna), 2009.

[74] TIOBE programming community index. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Accessed: 2016-05-11.

[75] JavaFX Frequently Asked Questions. `http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html#6`. Accessed: 2016-04-01.

[76] Ammar Rashid, William Y. C. Wang, and Dan Dorner. Gauging the Differences between Expectation and Systems Support: the Managerial Approach of Adaptive and Perfective Software Maintenance. In *Proc. of the 4th Int. Conference on Cooperation and Promotion of Information Resources in Science and Technology (COINFO)*, pages 45–50, 2009.

[77] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining Software Evolution to Predict Refactoring. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, pages 354–363. Ieee, sep 2007.

[78] W3C Web Storage Recommendation. `http://www.w3.org/TR/webstorage/#disk-space`. Accessed: 2016-04-01.

[79] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A Systematic Review of Software Maintainability Prediction and Metrics. In *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377, 2009.

[80] Shauvik Roy Choudhary. Cross-Platform Testing and Maintenance of Web and Mobile Applications. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 642–645, New York, NY, USA, 2014. ACM.

[81] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. Cross-Platform Feature Matching for Web Applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 82–92, New York, NY, USA, 2014. ACM.

[82] Winston W. Royce. Managing the Development of Large Software Systems. In *Proceedings of the 9th International Conference on Software Engineering*, pages 328–338. IEEE Press, aug 1987.

[83] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2 edition, 2005.

[84] Alexander Schatten, Stefan Biffl, Markus Demolsky, Erik Gostischa-Franta, Thomas Österreicher, and Dietmar Winkler. *Best Practice Software-Engineering*. Spektrum Akademischer Verlag, 2010.

[85] Norman F. Schneidewind and Christof Ebert. Preserve or Redesign Legacy Systems. *IEEE Software*, 15(4):14–17, 1998.

[86] Michael Seeboerger-Weichselbaum. *Programmieren mit Eclipse 3*. Mitp-Verlag, 2008.

[87] Alan Shalloway and James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley, 2004.

[88] Dag I. K. Sjøberg, Bente Anda, and Audris Mockus. Questioning Software Maintenance Metrics: A Comparative Case Study. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 107–110, 2012.

[89] Harry M. Sneed. Planning the Reengineering of Legacy Systems. *IEEE Software*, 12(1):24–34, 1995.

[90] ECMAScript Language Specification. `http://www.ecma-international.org/ecma-262/5.1/`. Accessed: 2016-04-01.

[91] Xmarks statistics of UMLet. `https://www.xmarks.com/site/www.umlet.com/`. Accessed: 2016-05-11.

[92] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.

[93] Unified Modeling Language (UML) version 2.4.1. `http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/`, 2011. Accessed: 2016-05-11.

[94] GWT Developer Guide Module XML. `http://www.gwtproject.org/doc/latest/DevGuideOrganizingProjects.html#DevGuideModuleXml`. Accessed: 2016-05-11.

[95] Jack Zhang, Shikhar Sagar, and Emad Shihab. The Evolution of Mobile Apps: An Exploratory Study. In *Proceedings of the International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2013, pages 1–8, New York, NY, USA, 2013. ACM.