Software Evolution of Legacy Systems
A Case Study of Soft-migration

Andreas Fürnweger, Martin Auer and Stefan Biffl
Vienna University of Technology, Inst. of Software Technology and Interactive Systems, Vienna, Austria

Keywords: Software Evolution, Migration, Legacy Systems.

Abstract: Software ages. It does so in relation to surrounding software components: as those are updated and modernized, static software becomes evermore outdated relative to them. Such legacy systems are either tried to be kept alive, or they are updated themselves, e.g., by re-factoring or porting—they evolve. Both approaches carry risks as well as maintenance cost profiles. In this paper, we give an overview of software evolution types and drivers; we outline costs and benefits of various evolution approaches; and we present tools and frameworks to facilitate so-called "soft" migration approaches. Finally, we describe a case study of an actual platform migration, along with pitfalls and lessons learned. This paper thus aims to give software practitioners—both resource-allocating managers and choice-weighing engineers—a general framework with which to tackle software evolution and a specific evolution case study in a frequently-encountered Java-based setup.

1 INTRODUCTION

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. 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, our paper first gives an overview on software evolution types, covering maintenance, reengineering, and whether to preserve or redesign legacy systems. We address software aging and its connection with maintainability. We look into different aspects of software maintenance and show 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. We also discuss program portability with a specific focus on porting source code.

We then outline costs and benefits of various evolution approaches. 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, we focus on various methods for "soft" migration approaches—those approaches aim to facilitate traditional migration methods like porting or rewriting code via support tools and frameworks. We especially concentrate on the Java programming language and present a specific variant of a soft-migration approach, which is using a Java-based program core with several platform-specific branches.

Finally, we describe a case study of an actual soft migration of the UML editor UMLet, which is currently available as a Swing-based Java program and an SWT-based Eclipse plugin, and which is ported to a web platform. We analyze some problems we encountered, and discuss the benefits and drawbacks of the suggested approach.

Fürnweger, A., Auer, M. and Biffl, S.
Software Evolution of Legacy Systems - A Case Study of Soft-migration.
In *Proceedings of the 18th International Conference on Enterprise Information Systems (ICEIS 2016) - Volume 1*, pages 413-424
ISBN: 978-989-758-187-8

2 RELATED WORK

(Mens and Demeyer, 2008) give an overview of trends in software evolution research and address the evolution of other 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, 2000), who also identify key problems and potential solution strategies.

Lehman classifies programs in terms of software evolution and also formulates laws of software evolution (Lehman, 1980; Lehman et al., 1997), which are, however, not considered universally valid (Herraiz et al., 2013). There are also many exploratory studies that try to analyze and understand software evolution based on specific software projects (Johari and Kaur, 2011; Businge et al., 2010; Zhang et al., 2013; Ratzinger et al., 2007; Kim et al., 2011). (Chaikalis and Chatzigeorgiou, 2015) develop a prediction model for software evolution and evaluate it against several open-source projects. (Benomar et al., 2015) present a technology to identify software evolution phases based on commits and releases.

The related topic of legacy systems is a bit ambiguous, due to differing definitions. It can describe a system that resists modification (Brodie and Stonebraker, 1995), a system without tests (Feathers, 2004), or even all software as soon as it has been written (Hunt and Thomas, 1999). A natural question regarding legacy systems is whether to preserve or redesign them. As this question is not easy to answer (Schneidewind and Ebert, 1998), the pros and cons of reengineering or preserving a system are to be compared thoroughly before making a decision (Sneed, 1995). In addition, it is possible to replace a system in stages to minimize the operational disruption of the system (Schneidewind and Ebert, 1998).

Even though the classic view of maintenance as the final life-cycle phase of software after delivery is still prevalent, it is a much broader topic, especially for programs which must constantly adapt to a changing environment. There are reports that the total maintenance costs are at least 40% of the initial development costs (Brooks Jr., 1995), 70% of the software budget (Harrison and Cook, 1990), and up to 90% of the total costs of the system (Rashid et al., 2009). As these numbers show, the topic of maintenance is crucial. (Lientz and Swanson, 1980) categorize maintenance activities into distinct classes. Several authors (Sjøberg et al., 2012; Riaz et al., 2009) propose maintainability metrics.

Finally, when migrating a system, a reengineering phase is almost always necessary. According to (Feathers, 2004), this phase should be accompanied by extensive testing to make sure the application behavior stays the same. (Fowler and Beck, 1999) list useful refactoring patterns, while (Feathers, 2004) stresses how legacy code can be made testable.

3 SOFTWARE EVOLUTION

This section outlines relevant disciplines and nomenclature related to software evolution.

3.1 Overview

(Lehman et al., 2000) divide the view on software evolution into two disciplines. The *scientific discipline* investigates the nature of software evolution and its properties, while the *engineering discipline* focuses on the practical aspects like *"theories, abstractions, languages, activities, methods and tools required to effectively and reliably evolve a software system."* (Lehman, 1980) classifies programs based on their relationship to the environment where they are executed. Lehman also formulates the eight *laws of software evolution*. Among those laws, two aspects are emphasized: *continuing change* (i.e., without adaption, software can become progressively less effective), and *increasing complexity* (i.e., as software evolves, its complexity tends to increase unless effort is spent to avoid that). According to (Herraiz et al., 2013), these laws have been proven in many cases, but they are not universally valid.

3.2 Legacy Systems

There are different definitions of what a legacy system or legacy code is. (Brodie and Stonebraker, 1995) describe it as *"a system which significantly resists modification and evolution."* (Feathers, 2004) defines it as code without tests, while (Hunt and Thomas, 1999) state that *"All software becomes legacy as soon as it's written."*

Preserve or Redesign Legacy Systems

According to (Schneidewind and Ebert, 1998), the question whether to preserve or redesign a legacy system is not easy to answer. In general, most organizations do not rush to replace legacy systems, because the successful operation of these systems is vital. 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, or applications.

An important aspect of this decision is that one does not have to 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 minimizes the disruption to the existing system and avoids replacing the existing system as a whole while it is operational (Schneidewind and Ebert, 1998).

(Sneed, 1995) remarks that reengineering is only one of many solutions to the typical maintenance problems with legacy systems. He also mentions that there must be a significant benefit, like cost reduction or added value, 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.

3.3 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." (Parnas, 1994)

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 (Monden et al., 2000).

3.4 Maintenance

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

Maintenance Effort and Costs

In large software codebases, the required maintenance effort is high. (Basili et al., 1996) 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 Jr., 1995) claims that the total maintenance costs of a widely used program are typically at least 40% of the initial development costs. (Rashid et al., 2009) 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, 1990), more than 70% of the software budget is spent on maintenance; 75% of software professionals are involved with maintenance. According to (Coleman et al., 1994), 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.

Maintenance Classes

(Lientz and Swanson, 1980) 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 (Sjøberg et al., 2012; Riaz et al., 2009). (Sjøberg et al., 2012) consider the overall system size to be the best predictor of maintainability.

3.5 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." (Chikofsky and Cross II, 1990)

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 reengineered."* (Demeyer et al., 2002)

(Feathers, 2004) 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., 2012) describe reengineering efforts to reduce the energy consumption of mobile devices.

3.6 Portability of Programs

Older high-level languages like C always aimed to be portable across systems, but often fall short, e.g., due to different APIs or system word size. To solve these problems, new languages were designed that run on

virtual machines. This was a huge step forward in terms of portability, as programs are compiled into an intermediate language that is runnable without modifications on any system with an implementation of the required virtual machine. Java is an example of such a language.

A similar approach is taken by web applications, which require a web browser instead of a virtual machine. The browser-based approach has other advantages like easy distribution. Web applications run on every platform with modern browser. Newer approaches based on system virtualization and containers (like Docker) address the need for better portability of whole subsystems without any restrictions on programming language or the used ecosystem.

Java Language and Platforms

Java is a programming language specifically designed for portability, achieved via virtual machines. They cover nearly all platforms, from smart cards and mobile phones to desktop and server environments. The most familiar non-official platform is probably Android, which supports large portions of the JavaSE API excluding graphical related portions such as Swing and AWT.

Other uses of the language are based on compilation of Java code to another programming language, such as GWT (Google Web Toolkit), which compiles from Java to JavaScript, or J2ObjC, which compiles from Java to Objective-C. Although most transpilers support a large part of the source language's features and API, certain features cannot be mapped to the target language (e.g., classes that are used for the Java GUI Framework Swing are not supported in GWT).

The main advantage of such a source-to-source compiler (also known as *transpiler*) is that there is no need for a Java Virtual Machine. This is especially important for the web platform, because even though browser-plugin-based Java Applets are possible, the plugin is based on the Netscape plugin API (NPAPI), which is not supported by mobile browsers. Furthermore, desktop browsers have also started to remove NPAPI support, e.g., the Chrome browser removed it on September 1, 2015.[1]

4 COSTS AND BENEFITS

This section discusses software evolution types and costs and benefits of migration/preservation.

[1] support.google.com/chrome/answer/6213033

4.1 Types of Software Evolution

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 bugfixes 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, ...).
   - 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.)

4.2 Software Evolution Criteria

After outlining some terminology and various aspects of software evolution, we can now summarize costs, risks, and benefits involved in migrating software to help determine the appropriate software evolution type.

Table 1: Comparison of costs/risks and benefits of preservation.

Preservation Risks	Preservation Benefits
Legacy systems are hard to maintain and change.	Stability (training, operations, ...) is preserved.
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.	Better predictability of overall system costs (if no major changes are required).
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, endusers might choose to use windows-based GUIs over their command-line-based ancestors.	Saved resources can be applied to keep the software alive with minor, and less dangerous, software evolution steps than outright migration, like partial reengineering, documentation via reverse-engineering, or virtualization.
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.	

Table 2: Comparison of costs/risks and benefits of migration.

Migration Risks	Migration Benefits
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.	Modern languages and related tools, a larger programmer base, faster hardware, ..., can reduce costs of new feature development, maintenance, and error fixing.
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.	Modern new software frameworks and libraries can improve the user experience, maintainability, and testability of the system.
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.	Better APIs can increase interoperability with modern software.
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.	New platforms (mobile, web, ...) can open up new markets and increase user acceptance.
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.	New code can be made more modular using object oriented design patterns, increasing its re-usability, and introduce automated tests (unit tests, integration tests, ...).
If the old system is not documented properly, knowledge that exists only implicitly within the program logic can get lost.	Vendor and platform dependency can be reduced (e.g., by removing libraries).

5 SOFT MIGRATION

Tools and frameworks can greatly facilitate software migrations; they allow for what we dub "soft" migrations. The next subsection gives a general overview on the variety of such migration assistance; the following one focuses on Java-based support.

5.1 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 with a browser plugin. Today, they are not very common anymore, because browsers started to remove the support for such plugins for security reasons (see section 3.6 about the removal of NPAPI support in browsers).

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, which transpiles from Java to JavaScript, or J2ObjC, which transpiles from Java to Objective-C.

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]

[2] supportline.microfocus.com/documentation/books/nx40/dijint.htm

[3] www.ikvm.net

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.

5.2 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, the Java Programming language can be run on nearly all commonly used platforms (any platform with a Java Virtual Machine (JVM) support, like Android, iOS, and GWT via transpilation). Unfortunately, not the full Java API is available on all of these 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.

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).

Supporting Technology

As mentioned in section 4.1, soft-migration relies on supporting tools. With Java, several such tools and frameworks are available:

- GWT is a Java to JavaScript compiler to facilitate migrations to web-based platforms.

- J2Objc is a Java to Objective-C compiler to port code to iOS.
- RoboVM is a Java ahead-of-time compiler and runtime, for iOS and OS X.

An alternative way to run Java applications within a web browser are Java Applets, but they depend on browser plugins, which are often limited in functionality for security reasons.

Steps

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. *Reengineer 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.

Creating New Software

In addition to the use case of migrating an existing application to a new platform, the idea of a shared Java core component can also be used when writing new software that should run on several platforms. Ray Cromwell gave a presentation at the GWT.create conference in January 2015 entitled *Google Inbox: Multi Platform Native Apps with GWT and J2ObjC*[4], where he explained details about how Google approached the development of their new product Inbox. He mentions that they share 60-70% of their code in a Java-based core component, which is (a) used as a Java Dependency for the Android application, (b) compiled to Objective-C (with J2Objc) for the iOS application, and (c) compiled to JavaScript (with GWT) for the web application.

Useful Tools

As mentioned, 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 codebases have no automated tests, therefore it is risky to refactor such code, because any change can easily break previously working features. Therefore it is 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.[5] It can be combined with Mockito[6] to create simple mocks of dependencies. Combined with Powermock[7], even static fields, final classes, and private methods can be mocked for tests.

  As legacy codebases often consist of tightly coupled components, it might be necessary to break those dependencies (see section 5.2) 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, 2004) 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[8] describes the pattern

[4] drive.google.com/file/d/0B3ktS-w9vr8IS2ZwQkw3WVRVeXc
[5] junit.org
[6] code.google.com/p/mockito
[7] www.powermock.org
[8] www.martinfowler.com/articles/injection.html

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. Examples for frameworks supporting dependency injection are Google Guice[9] or Spring.[10]

- *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,[11] PMD,[12] or Checkstyle.[13]
- *Build and Dependency Management.* Tools like Apache Maven[14] or Gradle[15] manage the dependencies of an application and its submodules. They also standardize several other aspects of an application like the directory structure and the build process. Their "convention over configuration" approach[16] also helps to familiarize new developers with a cade base, simply because of familiar project structure conventions.

6 EXEMPLARY MIGRATION

UMLet (Auer et al., 2009; Auer et al., 2003) is a UML tool in active development since 2001. It is referenced in 200+ publications, as well as 16+ books on software engineering. UMLet is the most favored plugin on the Eclipse Marketplace (Eclipse is the world-leading Java integrated development environment). In the 12 months leading up to August 1st 2015, more than 700.000 page views to UMLet's main web site have been recorded via Google Analytics.

UMLet uses a text-based approach of customizing UML elements (e.g., entering the line *fg=red* in the elements properties text block will color the background of the element red). Text without a specific meaning is simply printed, which is, e.g., a fast way to declare class methods.

To provide an exemplary application of the suggested soft-migration approach, UMLet gets migrated to a modern GWT-based web application that runs without browser plugins, while the Swing and Eclipse plugin versions are retained.

[9] github.com/google/guice
[10] spring.io
[11] findbugs.sourceforge.net
[12] pmd.sourceforge.net
[13] checkstyle.sourceforge.net
[14] maven.apache.org
[15] www.gradle.org
[16] softwareengineering.vazexqi.com/files/pattern.html

6.1 Legacy/Migration Criteria

Section 4.2 suggests two main decision drivers with the current UMLet codebase:

- The user base might move to new, web-based platforms, e.g., yUML[17], sketchboard[18], js-sequence-diagrams[19] or websequencediagrams[20].
- The current two-level platform (Java virtual machine on top of an OS) is not very future-proof:
  - Java often does not come pre-installed; it is not unlikely that future closed-source OS iterations further discourage Java deployments.
  - OS vendors like Apple 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).

These two criteria are the main drivers to use a migration approach with the goal of increasing the platform independence of UMLet.

6.2 Analysis

A first analysis shows that most of the applications code is tightly coupled with Swing classes. The main building blocks of the diagram (UML-Classes, -UseCases, -Relations, ...), which are called GridElement, all extend the Swing class JComponent.

The Eclipse plugin provides a small SWT-based wrapper around the Swing-based code to make it runnable in Eclipse. The parsing of the elements text is done within an overwritten JComponent.print() method, therefore there is no clear separation between parsing and drawing.

6.3 Reengineering

The goals of the UMLet reengineering are to:

- separate parsing and drawing of element properties;
- remove coupling between GridElement classes and Swing-specific classes;

[17] yuml.me/diagram/scruffy/class/draw
[18] sketchboard.me
[19] bramp.github.io/js-sequence-diagrams
[20] www.websequencediagrams.com

- introduce an abstraction layer with generic draw methods instead of directly relying on Swing Graphics objects;
- move GridElements to a separate module.

Even after the reengineering, there will be a relatively large portion of platform-specific code. E.g., the composition of the graphical user interface will still be platform-specific and must be implemented separately for GWT and Swing.

Based on this analysis, the given high-level architectural model can be retained with only minor differences on each platform (e.g., file-IO handling). The main restructuring of the model consists of a clear definition on how the properties of GridElements get parsed and drawn by each platform.

6.4 Implementation of Shared Codebase

As mentioned, the shared codebase mostly consists of the GridElements and the appropriate parsing and drawing logic.

New GridElements

The new GridElements have a unified syntax for the commands and therefore break backwards compatibility with some old diagrams. They are also reduced to a smaller set of customizable elements to avoid unnecessary element duplication.

Reusable Commands on Properties

The concept of element properties (and functions triggered by specific commands) is implemented using a separate parsing procedure, which is executed every time an element changes its properties or size. During this procedure, all possible commands for the specific element are checked and—if triggered—executed.

The main advantage of this approach is that these functions can be shared between elements. If two elements need to implement, e.g., the command *bg=red* to set the background color to red, they can refer to the same generic function. Changes like new features or bugfixes to such a function will therefore automatically be applied to all elements relying on them.

Common Drawing API

Platform-specific drawing logic is hidden behind a platform-independent API, which offers basic methods like drawLine(), drawRectangle(), printText(), as well as styling methods like setBackgroundColor() or setLineThickness(). Every platform has to implement this API and redirect the calls to the underlying graphical framework (e.g., Swing in JavaSE, or the HTML Canvas drawing methods in GWT).

Missing Basic Classes in GWT

As UMLet makes heavy use of geometric functionality, it needs classes such as Point, Line, Rectangle, ... Unfortunately, those classes are located in the AWT package and therefore not available on many platforms like GWT[21] or Android.[22] To circumvent this problem, alternative classes are created that are converted to platform-specific ones directly before drawing.

6.5 Web Implementation UMLetino

The web version of UMLet is called UMLetino and it transfers UMLet's minimalistic, text-based GUI approach to the web. The initial GUI mock was designed to look exactly like UMLet, but after further evaluation, it was apparent that a web application needs several adaptions. One difference, e.g., is the menu, which is a collapsible horizontal menu at the top border in most desktop applications, but a simple vertical menu on the left side for most web applications.

Another UI component that is different, because it is already embedded in the browser, is the tab-bar. An UMLetino-specific tab-bar below the browser tabs can be confusing and it does not prevent the user from opening multiple UMLetino tabs in the browser. It was therefore removed; users who want to work in parallel on several diagrams can rely on the native browser tabs instead.

Figure 1 shows the final, reengineered code structure in UML format.

Storing in Files or on the Web

UMLet stores diagrams in the file system. Web applications typically have limited access to it, therefore we have implemented several alternatives. Diagrams can be stored:

1. in the local storage of the browser (as a quick save/load while working on a diagram);
2. on the file system, with drag-and-drop-based import, and an export based on Data-URIs and the browser's save-as functionality;
3. on Dropbox[23] servers using the users' accounts.

[21] www.gwtproject.org/doc/latest/RefJreEmulation.html
[22] developer.android.com/reference/packages.html
[23] www.dropbox.com

The diagrams are stored using the XML-based UXF file format, which is also used in UMLet.

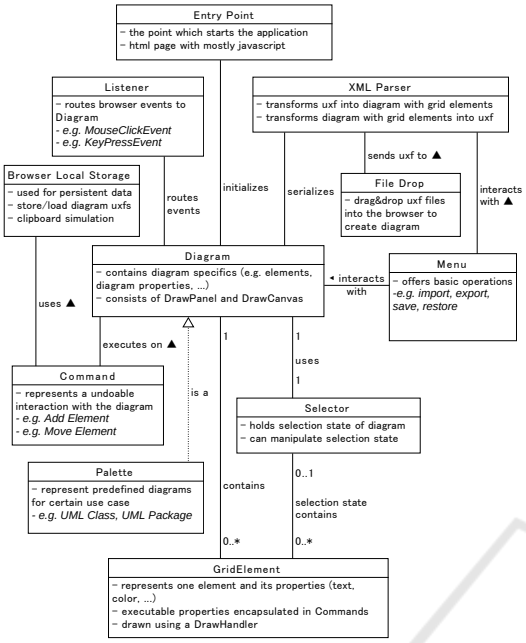

Figure 1: Reengineered code structure in UML format.

6.6 Code Base Analysis

Before Migration:

- 22,688 total (all in one project)

After Migration:

- 21,419 in Baselet (Standalone/UMLet specific)
- 8,915 in BaseletElements (shared)
- 3,135 in BaseletGWT (Web/UMLetino specific)
- 33,469 total

These numbers show that the web version consists of approx. 26% platform-specific code and 74% shared code.

The standalone version in comparison only consists of approx. 70% platform specific code and 30% shared code, but this is mostly due to the legacy support for the now deprecated OldGridElements. The old elements consist of roughly 5,600 LOC, so as soon as they are removed, approx. 36% of code will be shared.

Furthermore, there are some elements that have not been migrated to the shared codebase until now, due to their complexity (All in One Elements), or dependency on a Java Compiler during runtime (Custom Elements). They consist of roughly 4,000 LOC and will reduce the standalone specific code even more, while increasing the shared portion.

Although containing still much more specific code than the web version, the standalone project supports 3 different sub-platforms (Eclipse plugin, Swing standalone, and batch-mode) and therefore requires more code.

The overall duration of the migration was roughly 6 months; 2 developers in a remote-team setup spent an overall effort of 400 man-hours.

6.7 Lessons Learned

During UMLet's soft migration, we encountered several generic and specific issues worth mentioning:

- *Front-end code is often more platform-dependent and should be de-coupled from business logic.* There are several graphical libraries for JavaSE like AWT, Swing, or SWT. Android and GWT 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.
- *Choosing 3rd-party libraries creates dependencies and impacts the overall portability.* If a Java program should run on several platforms it must be verified that 3rd-party libraries work on all of them. In general, such libraries are only allowed to use Java classes that are supported by the platform specific API. In addition, GWT compiles Java source code to JavaScript, i.e., the 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 bytecode 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.

- *The documentation and tool support of GWT is very good, but the future is uncertain.* GWT is well documented and an Eclipse plugin eases development and testing. The GWT Dev Mode makes debugging within the IDE very convenient. Nevertheless, GWT Dev Mode is restricted to older browser versions (e.g., Firefox 26), because current browser versions have removed some required APIs (e.g., NPAPI). GWT offers the Super Dev Mode as alternative, but the Eclipse integration is only possible by using 3rd-party plugins like SDBG[24], and is less convenient.
- *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).

  Find UMLetino at www.umletino.com.

7 CONCLUSION

Software maintenance, aging, and evolution are often considered an afterthought. We hope to emphasize with this paper that software will inevitably age, and that this will surely have a non-trivial impact on its use and cost profile over time.

Within the general evolution process, planners and programmers can use a simple framework to help reach evolution decisions. A concrete instance of one application's soft migration hopefully helps to illustrate this. This should also underline how modern tools make software migration much more feasible. Future work should especially look a the recent container-based software deployment tools, especially with regard to outside interface dependencies. Of special interest are layers that interact with persistent data storage (typically databases). Another approach worth examining concerns GUI adaptability for various screen/input environments, especially as GUIs are notoriously tricky to migrate.

Finally, these considerations should not merely be applied "down the road," though this is still useful. Instead, the foreseeable eventual software evolution should be part of any decisions made during the software's *initial* design stages. Those are often crucial in making sure the software will age gracefully—and, ideally, never die.

[24] github.com/sdbg/sdbg

REFERENCES

Auer, M., Pölz, J., and Biffl, S. (2009). End-User Development in a Graphical User Interface Setting. In *Proc. 11th Int. Conf. on Enterprise Inf. Systems (ICEIS)*.

Auer, M., Tschurtschenthaler, T., and Biffl, S. (2003). A Flyweight UML Modelling Tool for Software Development in Heterogeneous Environments. In *Proc. 29th Conf. on EUROMICRO*.

Basili, V., Briand, L., Condon, S., Kim, Y.-M., Melo, W. L., and Valett, J. D. (1996). Understanding and Predicting the Process of Software Maintenance Releases. In *Proc. 18th Int. Conf. on Software Engineering (ICSE)*.

Bennett, K. H. and Rajlich, V. T. (2000). Software Maintenance and Evolution: A Roadmap. In *Proc. Conf. on The Future of Software Engineering (ICSE)*.

Benomar, O., Abdeen, H., Sahraoui, H., Poulin, P., and Saied, M. A. (2015). Detection of Software Evolution Phases Based on Development Activities. In *Proc. 23rd IEEE Int. Conf. on Program Comprehension (ICPC)*.

Brodie, M. L. and Stonebraker, M. (1995). *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann.

Brooks Jr., F. P. (1995). *The Mythical Man-Month*. Addison-Wesley.

Businge, J., Serebrenik, A., Brand, M. V. D., and van den Brand, M. (2010). An Empirical Study of the Evolution of Eclipse Third-party Plug-ins. In *Proc. Joint ERCIM WS on Software Evolution (EVOL) and Int. WS on Principles of Software Evolution (IWPSE)*.

Chaikalis, T. and Chatzigeorgiou, A. (2015). Forecasting Java Software Evolution Trends Employing Network Models. *IEEE Transactions on Software Engineering*, 41(6):582–602.

Chikofsky, E. J. and Cross II, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17.

Coleman, D., Ash, D., Lowther, B., and Oman, P. (1994). Using Metrics to Evaluate Software System Maintainability. *IEEE Computer*, 27(8):44–49.

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2002). *Object Oriented Reengineering Patterns*. Morgan Kaufmann.

Feathers, M. (2004). *Working Effectively with Legacy Code*. Prentice Hall.

Fowler, M. and Beck, K. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Gottschalk, M., Josefiok, M., Jelschen, J., and Winter, A. (2012). Removing Energy Code Smells with Reengineering Services. In *Beitragsband der 42. Jahrestagung der Gesellschaft für Informatik e.V. (GI)*.

Harrison, W. and Cook, C. (1990). Insights on Improving the Maintenance Process Through Software Measurement. In *Proc. Int. Conf. on Software Maintenance (ICSME)*.

Herraiz, I., Rodriguez, D., Robles, G., and Gonzalez-Barahona, J. M. (2013). The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review. *ACM Computing Surveys*, 46(2):1–28.

Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley.

Johari, K. and Kaur, A. (2011). Effect of Software Evolution on Software Metrics. *ACM SIGSOFT Software Engineering Notes*, 36(5):1–8.

Kim, M., Cai, D., and Kim, S. (2011). An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution. In *Proc. 33rd Int. Conf. on Software Engineering (ICSE)*.

Lehman, M. M. (1980). Programs, Life Cycles, and Laws of Software Evolution. In *Proc. IEEE*.

Lehman, M. M., Ramil, J. F., and Kahen, G. (2000). Evolution as a Noun and Evolution as a Verb. In *Proc. WS on Software and Organisation Co-evolution (SOCE)*.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and Laws of Software Evolution - The Nineties View. In *Proc. 4th Int. Symposium on Software Metrics (METRICS)*.

Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management*. Addison-Wesley.

Mens, T. and Demeyer, S. (2008). *Software Evolution*. Springer.

Monden, A., Sato, S.-i., Matsumoto, K.-i., and Inoue, K. (2000). 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.

Parnas, D. L. (1994). Software Aging. In *Proc. 16th Int. Conf. on Software Engineering (ICSE)*.

Rashid, A., Wang, W. Y. C., and Dorner, D. (2009). Gauging the Differences between Expectation and Systems Support: the Managerial Approach of Adaptive and Perfective Software Maintenance. In *Proc. 4th Int. Conf. on Cooperation and Promotion of Inf. Resources in Science and Techn. (COINFO)*.

Ratzinger, J., Sigmund, T., Vorburger, P., and Gall, H. (2007). Mining Software Evolution to Predict Refactoring. In *Proc. 1st Int. Symposium on Empirical Software Engineering and Measurement (ESEM)*.

Riaz, M., Mendes, E., and Tempero, E. (2009). A Systematic Review of Software Maintainability Prediction and Metrics. In *Proc. 3rd Int. Symp. on Empirical Software Engineering and Measurement (ESEM)*.

Schneidewind, N. F. and Ebert, C. (1998). Preserve or Redesign Legacy Systems. *IEEE Software*, 15(4):14–17.

Sjøberg, D. I. K., Anda, B., and Mockus, A. (2012). Questioning Software Maintenance Metrics: A Comparative Case Study. In *Proc. 6th Int. Symposium on Empirical Software Engineering and Measurement (ESEM)*.

Sneed, H. M. (1995). Planning the Reengineering of Legacy Systems. *IEEE Software*, 12(1):24–34.

Zhang, J., Sagar, S., and Shihab, E. (2013). The Evolution of Mobile Apps: An Exploratory Study. In *Proc. Int. WS on Software Development Lifecycle for Mobile (DeMobile)*.