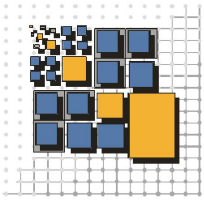

Otto-Friedrich-Universität Bamberg

Lehrstuhl für Praktische Informatik

Bachelor's Thesis

im Studiengang Software Systems Science
der Fakultät Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Zum Thema:

Migrating an Open Source Application to the Java 9 Module System

Vorgelegt von:

Florian Beetz

Themensteller:
Prof. Dr. Guido Wirtz

Abgabedatum:
19.08.2018

Contents

List of Figures

List of Tables

List of Listings

Abbreviations

ANT Another Neat Tool

API Application Programming Interface

BOM Bill Of Materials

CLI Command Line Interface

DAG Directed Acyclic Graph

DI Dependency Injection

EDT Event Dispatch Thread

GUI Graphical User Interface

IDE Integrated Development Environment

IoC Inversion of Control

JAR Java Archive

JDK Java Development Kit

JPMS Java Platform Module System

JSR Java Specification Request

LTS Long Term Support

POM Project Object Model

SDK Software Development Kit

SOA Service Oriented Architecture

SQL Structured Query Language

SLF4J Simple Logging Facade for Java

SOA Service Oriented Architecture

1 Introduction

In average 36% of development time of software systems is used on repaying technical debt – suboptimal decisions hindering software evolution and maintenance [1]. This is especially problematic for open source projects or software platforms, where compatibility with previous versions is usually highly valued to not frighten off users with the effort required to adapt to new versions of the system [2]. However, suboptimal decisions often become clear only later in the lifecycle of a software system and in order to simplify future development require to break backwards compatibility.

Recently, such a decision to break backwards compatibility in order to stay relevant was made by the developers of the Java language environment. With the release of Java 9 the fundamental way how code artifacts are organized changed with the introduction of a module system. Although several ways of migrating stepwise to the new system and utilizing the new features only partly, the new version of the language presents developers of the large ecosystem of Java libraries and applications wanting to migrate with problems.

While the topic of software maintenance and especially software migration is well studied [1, 3, 12, 13, 16] and also the migration of applications to different programming languages is covered [14], not much literature exists on the topic of migrating applications to newer versions of the same language. There are also several studies on the topic of API stability [2, 5], which is a big factor for backwards compatibility. Migration to Java 9 is mainly described in online documentation [26], text books [7, 10, 11] and online experience reports [30].

The goal of this thesis is to assess the difficulties that developers encounter when migrating applications from Java 8 to Java 9. To achieve this, the migration was performed exemplarily on the open source bibliography manager JabRef. In Section 2, first the topic of software migration in general is examined, then the advantages of and the way Java implements modules are analyzed, and the software JabRef is presented. Section 3 describes the iterative approach of the migration process applied to JabRef and the encountered problems. Then the software was also divided into several smaller modules, which is explained in Section 4. Finally, future work will be outlined in Section 5 and the thesis concludes in Section 6.

2 Background

2.1 Software Migration

Software migration is a part of software maintenance [5]. In general software maintenance can be classified as adaptive, corrective, perfective, and preventive maintenance tasks [12]. Within this classification, software migration is an adaptive maintenance task: Adapting existing software to a new environment or platform.

The need to perform a software migration usually arises, because the current platform or environment is obsolete or poorly supported, or from features available only in newer versions of a platform [12].

Software migration tasks can be grouped into three general classes:

- **Dialect conversion** is required when the underlying compiler technology changes to a new version of the compiler or a new compiler family [12]. Usually successive versions of compilers aim at being backwards compatible, even when new features were added, in large code bases, however, additional effort is required to use new versions of compilers.
- **API migration** is the process of changing a dependency on an external API to another one or a different version [12].

  Similar to other software, libraries that provide an API evolve over time, to introduce new features, fix bugs, and refactor source code [35]. APIs establish a contract with the clients, that rely on them, hence APIs should have a high stability to minimize effort for clients when updating to a newer version. However, not all changes in APIs are breaking the previously established contract, changes that do are referred to as *breaking changes*.

  Breaking changes mainly are modification or removal of existing API elements [2]. Adding new API elements are rarely braking changes.

  Usually libraries also contain code that is intended only for implementing the services offered by an API, but not for public consumption [5]. Many languages do not provide features to explicitly mark such elements as internal, but library authors rely solely on naming conventions, e.g. placing code in an `internal` namespace.
- **Language migration** is the decision to convert an existing program to a new language [12]. This is a risky type of migration, as it requires much effort to re-express source code in a different language.

The general approach for adaptive software maintenance consists of a sequence of steps as shown in Figure 1.

Applied to a migration process, the sequence of steps consists of first understanding the system. Then the changes in the new environment or platform need to be understood in order to understand the new requirements to the system. The next step is to develop a plan of how the new requirements can be implemented in the system. Next, the planned changes can be implemented, which may require debugging. Lastly, regression tests should be ran, to ensure that the system is still completely functional in the new environment.

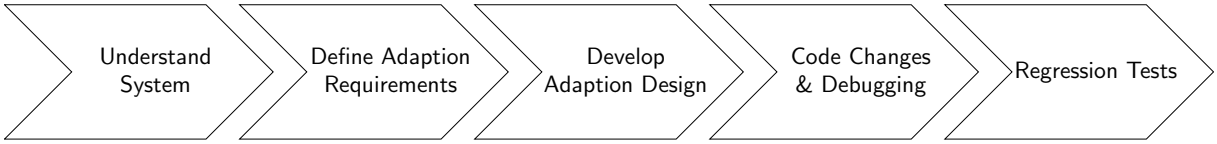

Figure 1: Activities of Adaptive Software Maintenance [16]

2.2 Java 9

The Java programming language was originally developed by Sun Microsystems under the name Oak in 1991 [29]. Oak was designed to be processor-independent and should be used in consumer electronics such as TV set-top boxes. In 1992 the industry had little interest in Oak, so the developers switched to bringing the language to the internet. The WebRunner browser was released in 1994, being the first browser supporting moving objects and dynamic executable content by supporting Oak. Due to trademark issues Oak was renamed to Java in 1995. In the same year the developers of the Netscape Navigator browser announced that they will include Java support. The first version of Java was then released in 1996. In 2009 Oracle acquired Sun and since then is the maintainer of the language [29].

Since its origin, Java has a very good history of being backwards compatible with previous releases [15, 20, 21, 22, 23, 24, 25, 28]. According to Oracle, incompatibilities usually occur only in rarely used edge- cases, or when new keywords were introduced in the language, such as `strictfp` in Java 2, `assert` in Java 4 and `enum` in Java 5, which subsequently can no longer be used as identifiers.

With Java 9 the *Java Platform Module System* (JPMS) – also popular under its working name *Jigsaw* – was introduced the the Java platform among other minor changes. JPMS adds *modules*, which are identifiable artifacts containing code, to the Java language [11]. The monolithic JDK itself was also split into smaller modules [4].

Since Java 9 the release cycle has also adapted to a faster pace [32]. Beginning with Java 9, a feature release will be published every six months and long term support (LTS) releases will be released every three years. Because of this, Java 9 is already superseded by Java 10 as of writing, and Java 11, the next LTS release, is expected to be released in September 2018.

2.2.1 Advantages of Modules

Before Java 9, artifacts were usually distributed as *Java archives* (JARs) [10]. Java has a concept of a classpath, which is a path in the file system Java searches for compiled code required at runtime or compilation. Figure 2 shows a schematic image of a classpath as it would be specified to Java. This classpath has 4 JAR files on it, each containing several packages. The white rectangles symbolize classes in the packages.

Before Java 9, the information of how packages and classes are organized was ignored by Java [10]. Java resolves classes on demand when they are first required. Figure 3 shows the information that is available to Java. All contents of the JARs on the classpath are seen as if it were only one artifact.

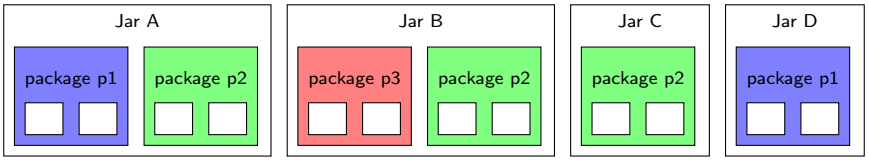

Figure 2: Unresolved Classpath before Java 9 [10]

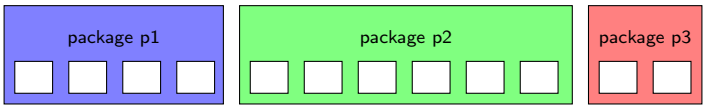

Figure 3: Resolved Classpath before Java 9 [10]

This way of handing loading of classes led to several problems: The first one being accessibility [10]. Every code artifact can essentially use code of every other artifacts, as long as classes or members are not restricted with the existing access modifiers (see Table 1). This can make dependency relations unclear for large code bases and violates strong encapsulation principles.

This becomes even more problematic with the second problem: If more than one type exists with the same fully qualified name, i.e. the package name and the type name is the same, the first one found is used [10]. This problem most often occurs when different versions of the same libraries are put on the classpath and is referred to as *JAR hell*. As classes are loaded lazily, those problems might not even be noticed on startup of an application, but only when it was running for some time and a class is used for the first time. Thus reliable configuration of the classpath is difficult and explains the rise of tools like Maven or Gradle, that standardize the process of obtaining dependencies and configuring Java to use them [10].

Table 1: Access modifiers and their associated scopes [11]

Access modifier	Class	Package	Subclass	Unrestricted
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
- *(default)*	✓	✓		
`private`	✓			

JPMS aims at exactly these needs of large Java applications: reliable configuration and strong encapsulation [4].

Modules have to explicitly declare which packages they make available to other modules and which modules they are dependent on [11]. Packages that are not exported by a module can not be used in other modules. This clearly separates public API from code that is intended for internal use only. Consequently internal code can also change freely without worrying about introducing breaking changes.

From the declaration of dependencies a so-called module graph (Figure 4) can be derived to identify dependencies of modules. The nodes of the graph represent the modules of the application, the dark blue edges represent the explicitly declared dependencies, while the light blue edges represent the implicit dependency of every module on the Java base module `java.base` [33].

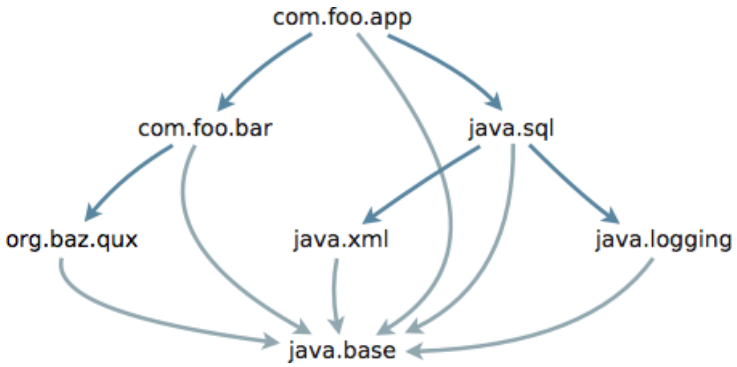

Figure 4: Module Graph [33]

Java 9 resolves modules every time before an application is compiled or executed [10]. Thus, it is possible to catch configuration errors like missing modules or multiple modules with the same name directly at startup.

Additionally due to the smaller modules, instead of one big runtime environment, Java 9 is better equipped to be run on devices for the Internet of Things (IoT) [7]. Those devices often have heavy restrictions on storage space and now do no longer have to store the full runtime environments, but only those parts that are required to run the respective application.

2.2.2 Modular Code before Java 9

Although modular code was not available to the Java at the language level, there are tools and libraries that provide similar functionality.

Many projects use a build tool to automate building source code, the most widely used ones in the Java ecosystem being ANT (Another Neat Tool), Apache Maven and Gradle [19]. The main goals of build tools are removing manual interaction, creating repeatable builds and making builds repeatable. Build tools let users express the tasks that need to be executed in order to build an artifact in an build file. Those tasks are ordered and thus are internally represented as a directed acyclic graph (DAG), to model dependencies on other tasks [17].

A key feature of build tools is the declaration of dependencies a project requires [19]. Figure 5 shows how the build tool Maven resolves dependencies: It first interprets the build script – in the case of Maven a Project Object Model (POM) file – checks the local cache if the dependency is already available on the local machine, and else downloads the required artifacts from a central Maven repository. Gradle implements similar dependency management functionality.

Since dependencies in the central Maven repository must also declare their dependencies, the dependencies of dependencies – so called *transitive dependencies* – can also be automatically be resolved [19].

Dependency management of Maven and Gradle is not limited to external libraries, but also supports decomposition of software into modules. While these modules can theoretically be accessed from every other module at runtime as described in Section 2.2.1, only the declared dependencies are put on the classpath on compile-time.

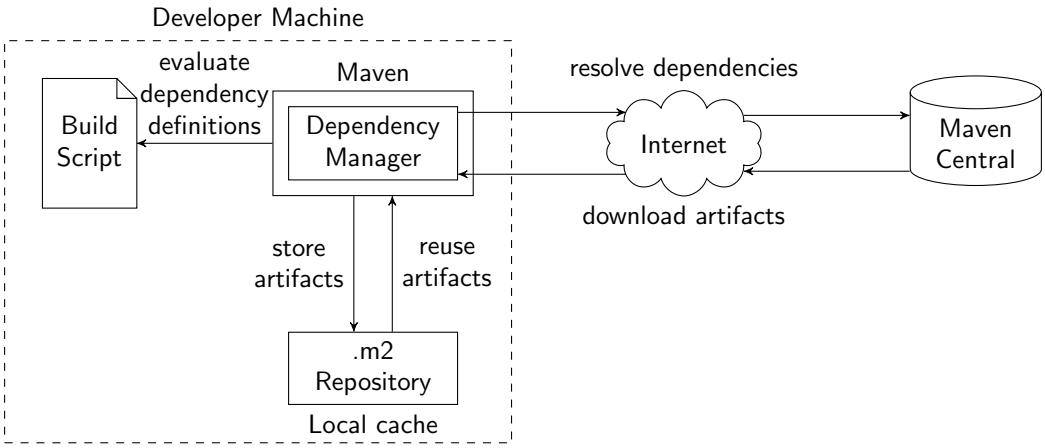

Figure 5: Maven's Interaction with Maven Central to resolve and download dependencies for a build [19]

The explicit declaration of dependencies, the ability to create a dependency graph by resolving external libraries and their transitive dependencies, and the possibility to define modules gives build tools similar functionality to JPMS. However, in contrast to JPMS build tools do not alter any behavior at runtime. While build tools handle the configuration of the classpath at compile-time, misconfiguration may still happen at runtime. Build tools – especially the resolving of dependencies – can be used in parallel with JPMS.

Another way of achieving modularity in Java before Java 9 is provided by the OSGi framework [34]. OSGi goes beyond the functionality of JPMS and provides in addition to a module system also a life-cycle management and a service registry.

The framework implements the module system by using custom classloaders – the mechanism Java uses to load classes – and metadata distributed in the manifest `META-INF/MANIFEST.MF` file of the module, which are called bundles by OSGi [6]. Listing 1 shows an example manifest of a bundle. Similar to JPMS the bundle declares its name, the packages it exports and the packages it will use from other bundles among other metadata.

Listing 1 OSGi Bundle Metadata [6]

```
Manifest-Version: 1.0
Created-By: 1.4 (Sun Microsystems Inc.)
Bundle-ManifestVersion: 2
Bundle-SymbolicName: org.foo.api
Bundle-Version: 1.0.0.SNAPSHOT
Bundle-Name: Simple Paint API
Export-Package: org.foo.api
Import-Package: javax.swing,org.foo.api
Bundle-License: http://www.opensource.org/licenses/apache2.0.php
```

In contrast to JPMS, OSGi bundles also have a certain life-cycle as shown in Figure 6 [6]. Bundles first need to be installed, they then are resolved. They then can be started and stopped at runtime. They also can be updated or even uninstalled at runtime.

This makes OSGi's module system much more dynamic than JPMS and explains why it is

commonly used as a plugin system to allow extending application's functionality [6].

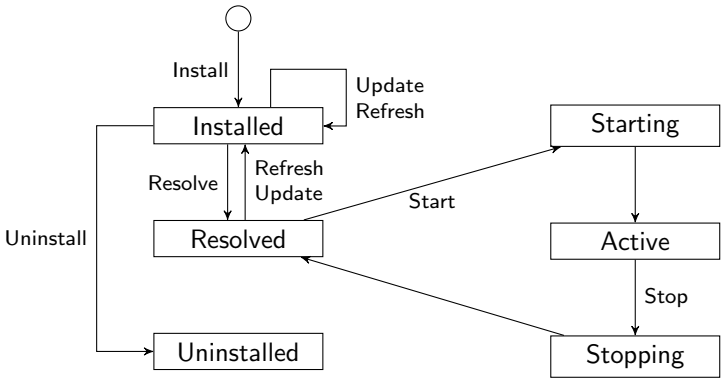

Figure 6: OSGi Bundle Life-Cycle [6]

The OSGi framework also provides an approach to implement a service oriented architecture (SOA) [6]. As shown in Figure 7, OSGi bundles can provide a description of the services they provide in the form of XML files. OSGi has a service registry, where the provided services are registered and can be looked up by other bundles to use them.

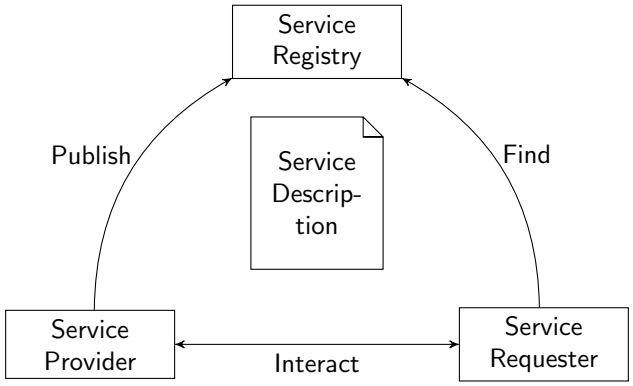

Figure 7: OSGi Service Registry [6]

In comparison to JPMS, OSGi provides much more elaborated features. However, JSR 376 specifying Java's module system states, that OSGi only can address the issue of reliable configuration, but not the issue of strong encapsulation as it is itself built on top of the Java platform [34]. Similar to the above mentioned build tools, OSGi can also be used together with Java 9.

2.2.3 JPMS' Modules

To make use of JPMS in Java 9 a module has to declare its public packages and its dependencies as mentioned in Section 2.2.1. This is done using a module descriptor, a file called `module-info.java` in the root package, that will be compiled as classes to a file called `module-info.class` [11]. Listing 2 shows an example of such a module descriptor.

In this example the module `com.company.module` is declared. It allows access from other modules to its packages `com.company.module.api`, `com.company.module.cli` and `com.company.module.gui`. Note however that although Java packages seem to appear

Listing 2 Module descriptor

```
module com.company.module {
    exports com.company.module.api;
    exports com.company.module.cli;
    exports com.company.module.gui;

    opens com.company.module.api.feature;

    requires org.thirdparty.module;
    requires transitive org.provider.othermodule;

    uses org.thirdparty.module.Service;

    provides com.company.module.api.Service
        with com.company.module.api.impl.ServiceImpl;
}
```

hierarchical, they are treated like regular identifiers, so if access to subpackages of any of the above packages should be allowed, they would have to be explicitly be exported [11].

The example module further "opens" the package `com.company.module.api.feature` to reflective access from other modules. Reflective access is not granted by default, even if a package is exported [11]. If a module relies heavily on reflection, it may also be declared as an `open module` to allow reflection into all its packages.

The module then declares its dependencies on `org.thirdparty.module` and `org.provider.othermodule`, with the second dependency being declared as transitive dependency. This means that the module also depends on all dependencies of `org.provider.othermodule`.

The last two declarations identify that the module uses a service `org.thirdparty.module.Service` provided by some other class and implements a service `com.company.module.api.Service` with the class `com.company.module.api.impl.ServiceImpl` for usage by other modules. This declaration of services was already a feature of Java before version 9, but relied on a configuration using text files. The declaration of provided and used services is JPMS form of dependency injection (DI), also known as the principle *Inversion of Control* (IoC) that allows hiding of implementation details [11].

As an alternative to the explicit declaration of a module descriptor, Java 9 also allows the usage of so called *automatic modules* [11]. Automatic Modules do not have a module descriptor, their name is derived from the attribute `Automatic-Module-Name` in the JAR manifest `META-INF/MANIFEST.MF` or from the name of the JAR file if that attribute is not present. An automatic module has some special characteristics: It `requires transitive` all other resolved modules, exports all its packages and reads the classpath. This version of module declaration is favorable if the module has to maintain backwards compatibility with previous Java versions. Especially library maintainers are often hesitant to migrate to the latest Java version to not lose their consumers using older versions.

The third variant of modules is the so-called *unnamed module* [7]. In contrast to explicit

modules and automatic modules, which are put on the *modulepath*, the unnamed module consists of all code that is put on the classpath. The unnamed module is treated like code before Java 9. Automatic modules can access code in the unnamed module, while explicit modules cannot.

2.2.4 Migrating to Modular Code

Oracle claims, that code that uses only official Java APIs should work without changes, but some third-party libraries may need to be upgraded [27]. However, in reality there are some more constraints of the module system that need consideration.

Firstly, due to the modularization of the JDK itself, internal APIs became unavailable [11]. Those classes were always meant to be used only internally by the JDK, but due to the missing access restrictions and missing alternatives, they have become adopted by some developers.

For widely used internal classes the module `jdk.unsupported` is provided, so that backwards compatibility for applications depending on them is ensured, however it is planned that those classes are replaced with supported alternatives in a future Java version [11].

While Java 9 still provides the possibility to explicitly make the internal APIs available with command line switches like `--add-exports`, the only long-term solution is to move away from those APIs and find supported replacement solutions [7]. Corresponding to that, the switch `--add-opens` exists for allowing reflection into packages of modules, that do not explicitly open packages.

The second restriction is that modules are no longer allowed to have split packages [11]. Split packages are packages with the same name, that are contained in two or more modules. Figure 8 shows two modules where both contain the packages `splitpackage` and `splitpackage.internal`.

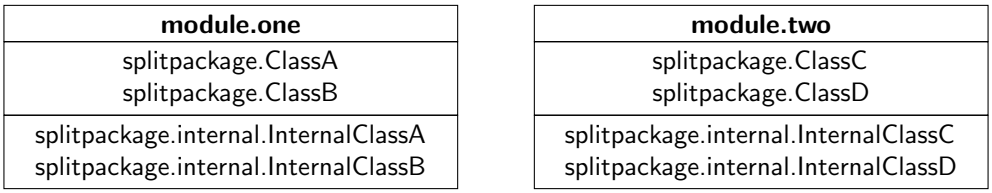

Figure 8: Split Packages across several Modules

If split packages were allowed, this would lead to inconsistencies in the encapsulation, as Java has a special visibility level for classes in the same package. It also could become unclear which class should be used, if two modules contain classes with the exact same fully qualified name.

Split packages across libraries cause runtime or compile-time errors such as the one shown in Listing 3.

Listing 3 Compiler error on split packages

```
error: the unnamed module reads package splitpackage from both module.one
and module.two
error: the unnamed module reads package splitpackage.internal from both
module.one and module.two
```

2.3 JabRef Bibliography Manager

JabRef is an open source bibliography reference manager using the standard LaTeX bibliography format BibTeX as its native file format. The project is hosted on GitHub[1] and currently has over 200 contributors and around 140,000 lines of Java code.

According to a survey carried out in 2015 across its users, JabRef is most commonly used by German, English and French speakers [9]. It is most commonly used in professional work, such as engineering and medicine and for studies, mostly from the field of natural sciences and formal sciences.

JabRef is built using a layered architecture as shown in Figure 9 [8]. The shown components only depend on components lower in the figure. The base of the architecture is the *Model* component, that encapsulates the entities used in the application. Building on top of that component is the *Logic* component, that contains all business logic. The *Preferences* component provides the functionality to load and store user defined settings. JabRef's command line interface is encapsulated in the *CLI* component and the top layer is constituted by the graphical user interface in the *GUI* component. Additionally, there exist some additional global classes, that may be used anywhere in the application.

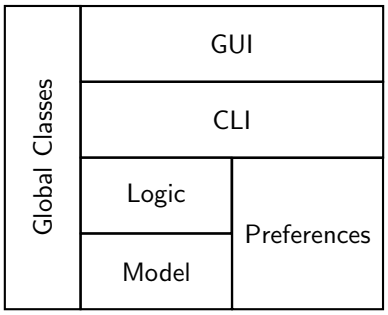

Figure 9: High-Level Architecture of JabRef

The communication between the components of JabRef is implemented using an event bus, that allows publishing events and registering listeners for events. This allows to react upon changes in the core and still react in the upper layers, while keeping the components clearly separated.

The source code of JabRef is build using the tool Gradle. Gradle automates repeated tasks such as compilation of the source code, building release distributions, resolving correct versions of dependencies, running tests and generation of source code with parser generators.

[1] `https://github.com/JabRef/jabref`

3 Migrating JabRef to Java 9

The following section covers the process of migrating JabRef from Java 8 to Java 9. Due to the open source nature of JabRef development of the project continued during the migration phase. Therefore the migration technique as shown in Section 2.1 was applied in an iterative approach and changes to the current version were continuously synchronized to the Java 9 version.

Figure 10 shows the general approach of the migration. The approach is substantially different depending on whether an issue is located in an external dependency or in JabRef's codebase. When the issues were fixed in a future version of a third-party library, the solution simply consists of upgrading said dependency, otherwise the issues were reported to the maintainers of the respective libraries or a code contribution to their projects was created.

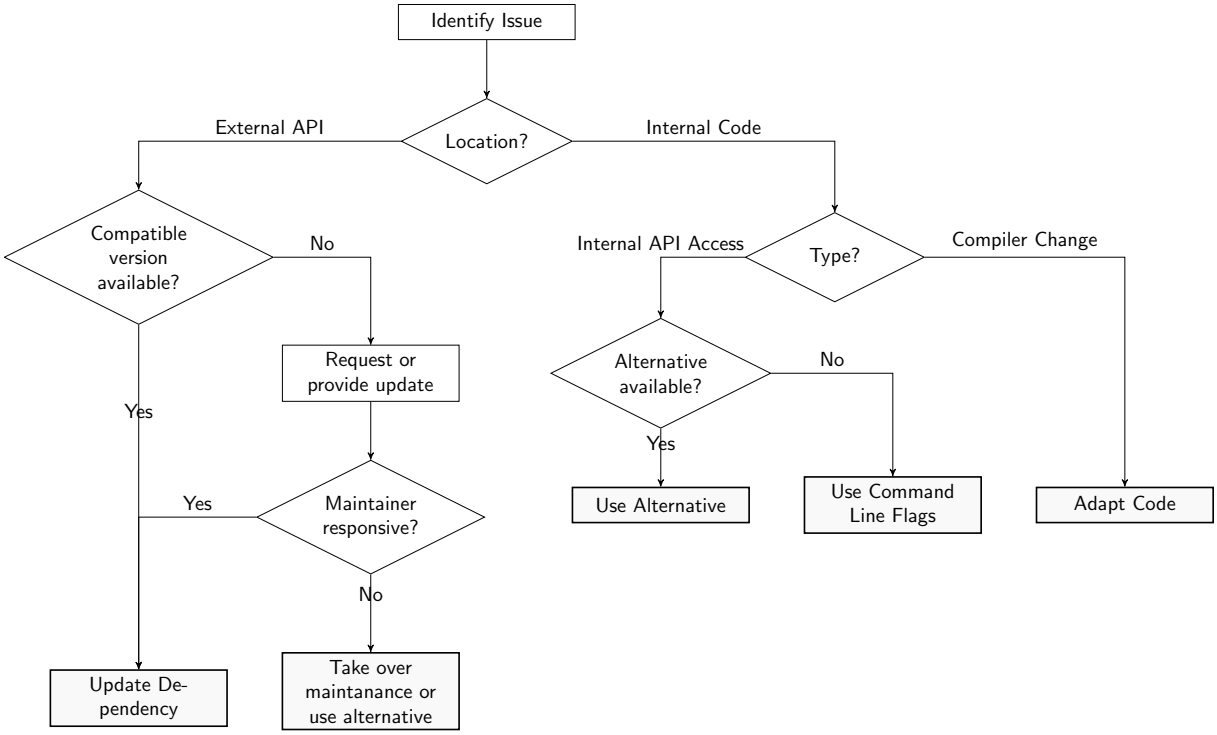

Figure 10: General Approach of the Migration

Issues in JabRef internal code can be classified into access to now internal API and changes in the Java compiler. The only sustainable solution to those problems is migrating away from the API and finding a supported replacement. Changes in the compiler are usually only minor, but require adaption of the code.

3.1 Compile-Time Compatibility

In the first iteration the focus lay on ensuring compile-time compatibility with Java 9. Incompatible parts of JabRef were temporarily removed.

A number of external libraries were incompatible with Java 9 and also had to be removed. The incompatibilities of these libraries can be categorized into the following categories.

First, four libraries exported the same packages, resulting in a split package as explained in Section 2.2.4. These libraries were the popular utility library *Google Guava*, the SDK of the office suite *LibreOffice*, Microsoft's monitoring service *ApplicationInsights* and *ArchUnit*, a test framework to check for architecture constraints.

While Google Guava did not actually contain a split package, it had a dependency on an unofficial implementation of Java annotations as specified by the Java Specification Request (JSR) 305, that aims at assisting tools to find software defects by providing annotations such as `@NonNull` [31]. However, this dependency was optional and thus not required at runtime, so the solution was to explicitly exclude it in the Gradle build script as shown in Listing 4.

Listing 4 Exclusion of the JSR 305 dependency

```
configurations {
    // [...]
    compile {
        exclude group: 'com.google.code.findbugs', module: 'jsr305'
    }
}
```

For ArchUnit a development version was already available, so it could simply be updated. The LibreOffice SDK and Microsoft ApplicationInsights were incompatible with Java 9, so for the first iteration, they were temporarily removed and the features of JabRef depending on them disabled.

Second, the GUI libraries *Spin* and *ControlsFX* and JabRef itself used internal APIs in the JDK, that were no longer accessible. For the first iteration, the solution was to simply use the flags mentioned in Section 2.2.4 to allow the access to those libraries. Listing 5 shows the command line arguments required to run JabRef in the first iteration.

Listing 5 Command Line Arguments Required to Run JabRef in Iteration 1

```
java \
    --illegal-access=debug \
    --add-opens javafx.swing/javafx.embed.swing=org.jabref \
    --add-opens java.desktop/java.awt=spin \
    --add-opens javafx.controls/javafx.scene.control=org.jabref \
    --add-exports javafx.base/com.sun.javafx.runtime=controlsfx \
    -p . \
    -m org.jabref/org.jabref.JabRefMain
```

The third problem for compile-time compatibility were the module names of some dependencies. As mentioned in Section 2.2.4 Java first searches for a module descriptor, if it can not be found the `Automatic-Module-Name` attribute in the JAR manifest is consulted and if that is not present, Java derives a module name from the file name of the JAR file. However, module names underlie the same restrictions as Java packages, so they may contain dots, but each segment between two dots must be a valid Java identifier.

This was a problem for Scala dependencies that follow the default Scala naming scheme consisting of the name of the project followed by an underscore followed by the Scala version. So an artifact with the name `latex2unicode` for Scala 2.11 results in an artifact with the name `latex2unicode_2.12.jar`. For this, Java 9 derives the module name `latex2unicode.2.12`, which is an invalid module name as the 2 directly follows a dot. For the first iteration the Scala dependency *latex2unicode*, which JabRef uses to resolve LaTeX commands to plain text, that had again dependencies on the Scala libraries *fastparse* and *Sourcecode*, was temporarily removed.

Fourth, there were minor incompatibilities with the new compiler regarding the use of Java generics. The method `children()` in Listing 6 returns an object of the type `Enumeration<TreeNode>`, before Java 9 however, it could be assigned to a variable of the type `Enumeration<CheckableTreeNode>` where `CheckableTreeNode` inherits `TreeNode`.

Listing 6 Use of Generics before Java 9

```
Enumeration<CheckableTreeNode> tmpChildren = this.children();
for (CheckableTreeNode child : Collections.list(tmpChildren)) {
    child.setSelected(bSelected);
}
```

This direct conversion is no longer possible in Java 9. The returned object of `children()` is assigned to a variable of the correct type `Enumeration<TreeNode>` in Listing 7 and cast to the type `CheckableTreeNode` on usage.

Listing 7 Use of Generics after Java 9

```
Enumeration<TreeNode> tmpChildren = this.children();
for (TreeNode child : Collections.list(tmpChildren)) {
    ((CheckableTreeNode) child).setSelected(bSelected);
}
```

Several instances of that problem were found throughout JabRef's source code, however no documentation could be found that explains this change in the Java language.

Lastly, as a result of the first iteration also a module descriptor was created for JabRef (see Section 2.2.3). While it would have been possible to make JabRef an automatic module, instead of an explicit one, there were already efforts for creating a descriptor due to the open source nature of JabRef.

Listing 8 shows an excerpt from the module descriptor. The module was declared as open module to allow all internal access into JabRef, because the architecture of JabRef (see Section 2.3) is based around an event bus provided by the Google Guava library, which makes extensive use of reflection.

Listing 8 JabRef module

```
open module org.jabref {
    exports org.jabref;

    exports org.jabref.gui;
    // [...]

    // SQL
    requires java.sql;
    requires postgresql;

    // JavaFX
    requires javafx.graphics;
    requires javafx.swing;

    // [...]

    provides com.airhacks.afterburner.views.ResourceLocator
        with org.jabref.gui.util.JabRefResourceLocator;

    provides com.airhacks.afterburner.injection.PresenterFactory
        with org.jabref.gui.DefaultInjector;

    // [...]
}
```

3.2 Upgrading Libraries

In the second iteration the focus lay on updating the libraries removed in iteration one to versions that are compatible with Java 9. Not much work was done on JabRef itself, but getting in contact with library maintainers and participation in their open source projects was the main objective.

3.2.1 LibreOffice

JabRef uses the LibreOffice SDK to insert citations and references into LibreOffice documents. However, the SDK consists of multiple artifacts all exporting the same package `com.sun.star`, so they are incompatible with JPMS due to a split package (see Section 3.1). Thus the complete SDK and JabRef's functionality to interface with LibreOffice was temporarily removed.

Possible long-term solutions include bundling all artifacts as one artifact, so the LibreOffice SDK is no longer modular, but requires consumers to load all of it. The problem of the split package however would be solved, as the SDK is then only one module to export the package. Another solution could be to rename the packages contained in each artifact, this however would break backwards-compatibility of the SDK.

The issue was reported to the Document Foundation, the maintainer of LibreOffice[2]. Table 2 shows the development on the bug report.

Table 2: Timeline of the bug report of the split package in LibreOffice

Date	Action
2018-04-29	Bug Report Created
2018-06-07	Bug Confirmed by another user

As the developers of LibreOffice were unresponsive to the bug report as of writing, a possible workaround to the problem would be to manually repackage the artifacts to a single one as proposed above. However, doing this without support of the original developers would complicate the build process of JabRef, because the patched artifact would need to be shipped with the source code instead of downloading the dependencies from a central Maven repository as it is done for other dependencies.

3.2.2 Latex2Unicode

Latex2Unicode is written in Scala and has dependencies on three other Scala libraries. As briefly shown in Section 3.1 Scala's default naming scheme generates invalid automatic module names, so latex2unicode had to be temporarily removed in iteration one.

Since Scala does not yet support JPMS, the solution of this problem is to explicitly provide an `Automatic-Module-Name` attribute in the libraries manifest (see Section 2.2.3). This was proposed to the library maintainer of latex2unicode[3] and to the maintainers of the dependent libraries fastparse[4] and sourcecode[5] in the form of code contributions – so called pull requests – to their libraries.

Table 3 shows the timeline of the bug report for latex2unicode.

Table 3: Timeline of the bug report for latex2unicode

Date	Action
2018-04-28	Bug report created
2018-05-18	Code contribution provided
2018-05-29	Proposed fix accepted by library maintainer
2018-xx-xx	Version including fix published

The maintainer of the dependencies fastparse and sourcecode remained unresponsive to the proposed fixes both provided on 2018-05-18 as of writing. Possible solutions to work around the problem include providing a manually patched version of the libraries or using a service such as Jitpack[6] to build the versions including provided code contribution. Jitpack allows

[2] `https://bugs.documentfoundation.org//show_bug.cgi?id=117331`
[3] `https://github.com/tomtung/latex2unicode/pull/11`
[4] `https://github.com/lihaoyi/fastparse/pull/185`
[5] `https://github.com/lihaoyi/sourcecode/pull/49`
[6] `https://jitpack.io`

developers to publish versions of their libraries directly from a Git repository without additional configuration.

3.2.3 Microsoft ApplicationInsights

ApplicationInsights follows the practice to distribute so called fat JARs – Java artifacts including all required dependencies – but additionally relocate the packages of dependencies under their own package prefix. So their dependency on Google Guava using the package `com.google.common` is distributed in ApplicationInsights as `com.microsoft.applicationinsights.core.dependencies.googlecommon`.

However, Guava also exports the package `com.google.thirdparty`. This package was not correctly relocated in ApplicationInsights, causing a split package conflict with JabRef's dependency on Google Guava.

The problem was reported to the library maintainers of ApplicationInsights[7]. Table 4 shows the timeline of the bug report.

Table 4: Timeline of the bug report of missed package relocation in Google Guava

Date	Action
2018-05-05	Bug Report Created
2018-06-08	Bug fixed by maintainers of the library
2018-xx-xx	Version including fix published

3.2.4 JavaFxSVG

JavaFxSVG is a library that allows the GUI framework JavaFX to display SVG graphics. The library exports the package `org.w3c.dom` which conflicts with APIs provided directly from the JDK. However, this functionality was used at only one occasion – to display the JabRef logo in an About dialog – so in agreement with the JabRef developers the library was removed and replaced with JavaFX's native capabilities.

Figure 11: JabRef logo

JavaFX does not support the full set of features of the SVG definition, but it has support for its so called paths specifying vertices of geometric shapes. JabRef's logo consists of six such paths as shown in Figure 11. The solution was to overlay the paths in order to recreate the complete image (see Listing 9).

[7] `https://github.com/Microsoft/ApplicationInsights-Java/issues/661`

Listing 9 Rendering of JabRef logo with JavaFX

```
<StackPane onMouseClicked="#openJabrefWebsite" scaleX="0.6" scaleY="0.6"
    prefWidth="140" prefHeight="140" BorderPane.alignment="CENTER">
    <!-- SVGPaths need to be wrapped in a Pane to get them to the same
    size -->
    <Pane prefHeight="350" prefWidth="350" styleClass="logo-pane">
        <SVGPath content="M97.2 87.1C93.2 33.8 18.4 14.6 18.2 ..." />
    </Pane>
    <Pane prefHeight="350" prefWidth="350" styleClass="logo-pane">
        <SVGPath content="M96.2 61.2C92.8 19.2 35.1 0.4 35 ..." />
    </Pane>
    <!-- ... -->
</StackPane>
```

3.3 Reworking JabRef's Threading Model

The third iteration of migrating JabRef to Java 9 consisted in reworking parts of its threading model. The rework was required because JabRef used the library "Spin" to simplify interaction of the GUI with long-running background tasks. Spin provides utilities to load off time-intensive operations to a separate thread, so that the GUI stays responsive to user input.

This is done by creating proxy objects that run their operations on a separate thread, but wait until their execution has finished, while keeping the GUI thread responsive (see Figure 12).

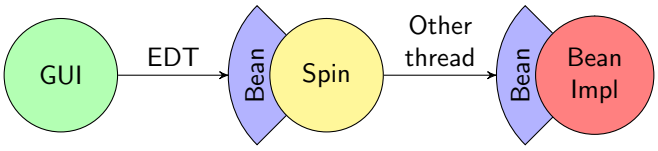

Figure 12: Model of Spin [18]

However, the JabRef developers are in the process of migrating from the Swing GUI framework, that Spin was written for, to the newer GUI framework JavaFX. This migration process already lead to some threading issues, because often GUI frameworks restrict programmatic interactions with GUI components to be only allowed on the GUI thread – sometimes also called event dispatch thread (EDT). This migration process made the usage of Spin obsolete, as it does not work with JavaFX.

The solution to this problem was to adapt the approach of JabRef, that was already employed in parts of the application. JabRef uses an callback based approach partly provided by the JavaFX framework itself. This approach provides a class `BackgroundTask` that wraps time consuming operations and provides means to specify callbacks that are executed on the GUI thread once the operations succeeds, fails or either of the two.

Listing 10 shows how background tasks are used in JabRef. The method `verifyDuplicates` is executed on a thread of the `Globals.TASK_EXECUTOR` executor service. When the verification of duplicates succeeds the method `handleDuplicates` is called on the JavaFX

Listing 10 Usage of background tasks

```
BackgroundTask.wrap(this::verifyDuplicates)
               .onSuccess(this::handleDuplicates)
               .executeWith(Globals.TASK_EXECUTOR);
```

GUI thread, failures are not handled in this case.

4 Modularizing JabRef

After JabRef was running with Java 9, the next goal was to modularize the application in order to reinforce the architectural rules as shown in Section 2.3, but also to extract useful libraries for other applications. In the past there already efforts to extract libraries from JabRef using the build tool Gradle's support for modules[8]. Using this approach JabRef would not produce one monolithic JAR artifact, but several smaller JAR artifacts depending on each other. The problems that JPMS addresses (see Section 2.2.1), however, would not be addressed using this approach. The changes were discarded due to the release of Java 9.

In order modularize the application with JPMS, an approach as shown in Figure 13 was applied iteratively.

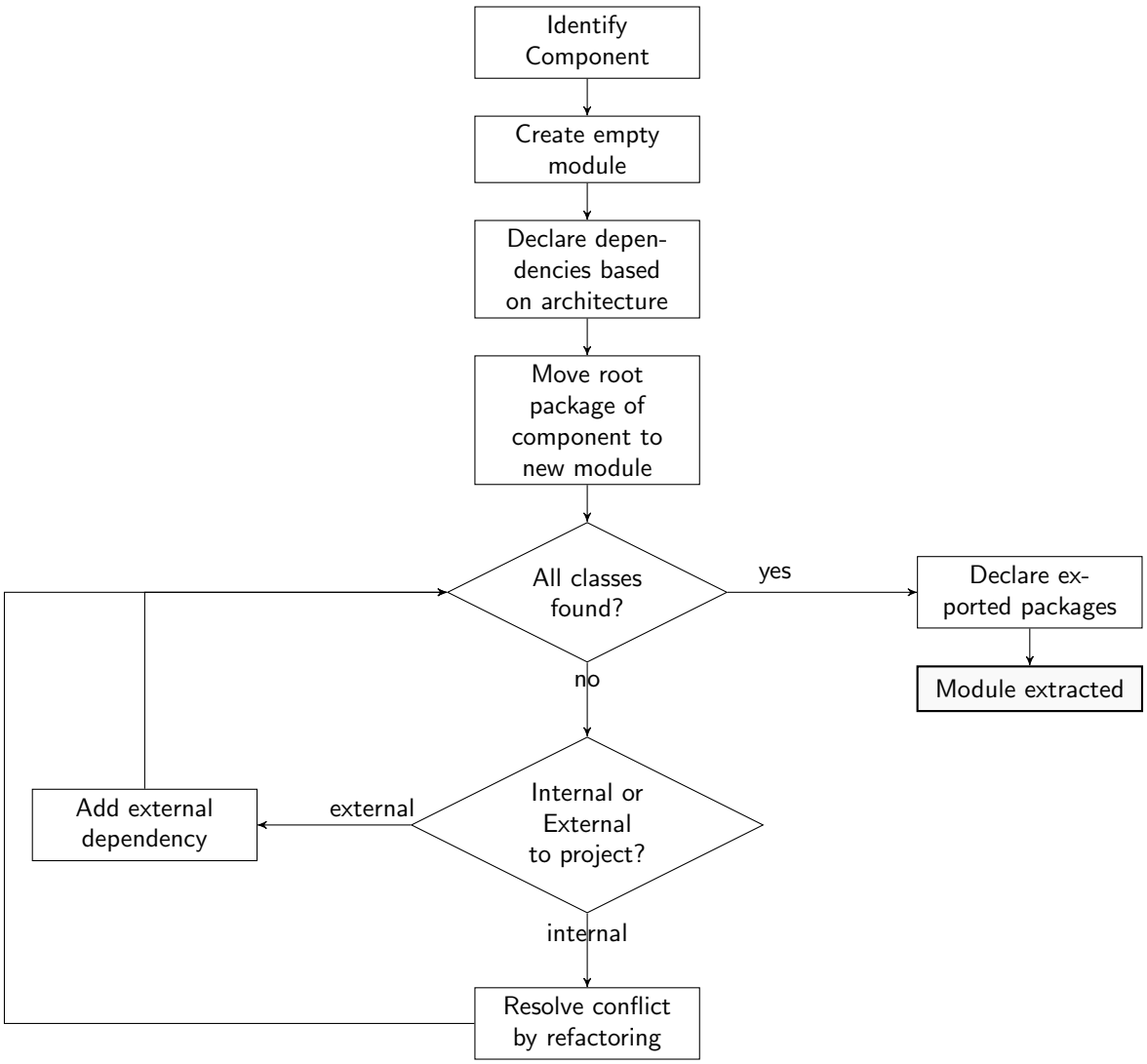

Figure 13: Approach of Modularizing an Application with JPMS

First a component was chosen and an empty module was created for it. Then the dependencies were added according to the planned architecture. Then the packages where the component resides in was moved to the new module. In order to find missing

[8] https://github.com/JabRef/jabref/pull/3704

dependencies, the new module was repeatedly compiled. By analyzing the compiler errors, missing dependencies could be found. Dependencies on external libraries could easily be added to the build script and the module descriptor. Internal conflicts required appropriate refactoring according to the problems at hand. Once the new module compiles without errors, the packages that should be exported could be declared. Lastly, the application with the extracted module was ran to ensure the functionality of the application.

The modularization was performed with a bottom-up approach. First the components with no dependencies on other components were extracted, then the components with only dependencies on already modularized components were extracted and so forth. This was done to avoid circular dependencies, which are disallowed by JPMS [11].

4.1 Handling Illegal Dependencies

The first problem encountered in the modularization were violations of the architecture, so parts of a component had dependencies on components, where – according to the architecture – this dependency should not exist.

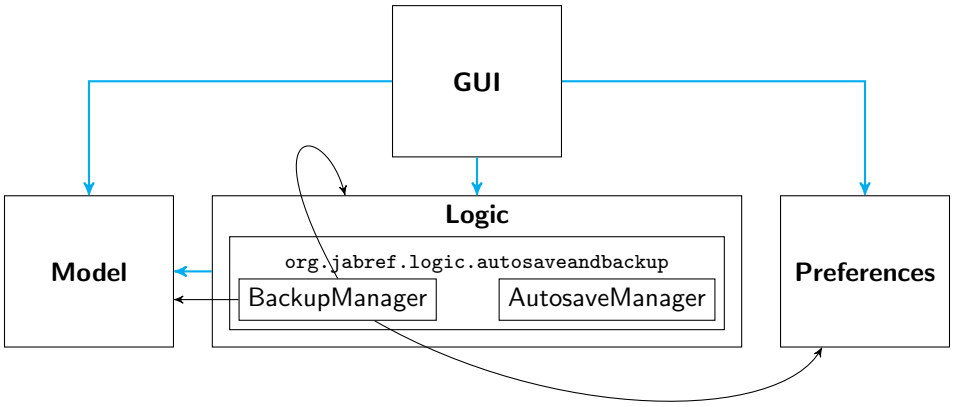

Figure 14: Illegal Dependency of `BackupManager`

Figure 14 shows the dependencies of the class `BackupManager` in the package `org.jabref.logic.autosaveandbackup` of the Logic component on classes of the components Model, Logic and Preferences. However, the Logic component only depends on the Model component, thus the classes in the Preferences component become unavailable after modularization.

There are multiple scenarios how this conflict could be solved: First, the architecture could be simply changed, so that the Logic module depends on the Preferences component. If many instances of the same problem exist in the code base, this might be a practical solution. In JabRef's case however, this was the only occurrence of this problem, so changing the the architecture of the application to make one component available is not the ideal solution.

Second, the dependency on the Preferences module could be removed. Depending on the type of the dependency this may be a viable solution, for example if alternatives for the used functionalities are available. In this case however, removing the dependency would have lead to code duplication, which is unfavorable for enabling good software maintainability.

Third, the components Logic and Preferences could be joined in one module. This would weaken the encapsulation of the components, but especially if there exist many circular dependencies between the components, this might be the only acceptable solution. In JabRef's case there were no circular dependencies between the modules, so the modules were not joined.

Lastly, the parts of the component causing the conflict could be moved up in the dependency hierarchy to the next component that depends on all required components. This solution has the downside, that the API of the affected components become scattered across modules, but depending on the use-case this might be acceptable. Another thing to keep in mind, is that Java 9 does no longer allow split packages, so either the whole package needs to be moved, or the package needs to be split into two different packages. For JabRef, this solution was pursued since the affected parts of the API are only a small part of the component and the solution was already implemented previously[9].

A slightly more complex conflict occurred in the class `OpenDatabase` as shown in Figure 15. Here, the action responsible for opening literature databases also performs several migrations on older database versions. However, the migrations located in `org.jabref.migrations` are considered global classes in the architecture. Some of the migrations also require user interaction, so they depend on the GUI component, while others do not and only depend on the Logic and Model component[10].

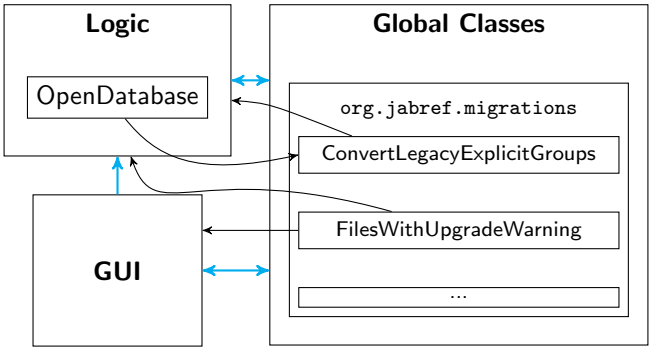

Figure 15: Architecture Conflict in `OpenDatabase`

The possibilities to solve this conflict are roughly the same as before, however the change of architecture and the removal of the dependency was not favorable for the same reasons as before. To solve this conflict a combination of joining the components and moving the conflicting classes up in the hierarchy was chosen. First the package `org.jabref.migrations` was split into the packages `org.jabref.logic.migrations` containing the classes that require no user interaction and `org.jabref.gui.migratons` for the classes that do. The `logic` part of the migrations was then moved to the Logic component and the `gui` part to the GUI component.

Moving the `OpenDatabase` action up in the dependency hierarchy to the GUI component in this case would have made no sense, as it would have required to move large parts of the Logic module along with it. Likewise, moving the whole `migrations` package to the Logic module would have required to move large parts of the GUI component along. Both

[9] `https://github.com/jabref/jabref/tree/multi-module-build/org.jabref.gui/src/main/java/org/jabref/logic/autosaveandbackup`

[10] The dependencies on the Model component are not shown in Figure 15.

would have reduced the encapsulation of the modules and scattered the components across both modules.

4.2 Tool Support for Modularization

Another problem when modularizing JabRef was the lacking support of the used tools for modularized Java code. As mentioned in Section 2.3, JabRef is built using the build tool Gradle. While Gradle has an experimental plugin called *experimental-jigsaw* providing support for JPMS[11], this plugin but according to the authors currently is "not very sophisticated or particularly well-tested". Also development on this project is stale for over a year as of writing.

To build JabRef a derivative of the above plugin called *chainsaw* is used. Chainsaw[12] is an open-source project and has more sophisticated features than experimental-jigsaw, such as supporting popular unit test frameworks, patching modules and providing support for the flags `--add-exports` and `--add-opens` as described in Section 2.2.4. However, chainsaw does not support building projects consisting of multiple modules. This issue is reported to the maintainer of the project[13], but is unresolved as of writing.

This issue can be resolved by applying a workaround as shown in Listing 11. With this workaround, the dependencies of the task `compileJava` are explicitly set to the `assemble` task, that builds a JAR file, of the modules `jabref-model` and `jabref-logic`.

Listing 11 Workaround for Multi-Module Builds using Chainsaw

```
compileJava.dependsOn ":jabref-model:assemble"
compileJava.dependsOn ":jabref-logic:assemble"
```

Another problem in the tool support for JPMS is Gradle itself. Gradle supports running automated unit tests which is executed by default on every build. However, there is an unfixed bug in Gradle[14] that causes errors when the wide adopted logging library *SLF4J* (Simple Logging Facade for Java) is used in the tests with Java 9.

Listing 12 Gradle causes Errors when SLF4J is used

```
SLF4J: No SLF4J providers were found.
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#noProviders for further
details.
java.lang.ClassCastException: org.slf4j/org.slf4j.helpers.NOPLoggerFactory
cannot be cast to org.gradle.internal.logging.slf4j.
OutputEventListenerBackedLoggerContext
```

[11] https://github.com/gradle/gradle-java-modules
[12] https://github.com/zyxist/chainsaw
[13] https://github.com/zyxist/chainsaw/issues/34
[14] https://github.com/gradle/gradle/issues/4974

This is a major impediment for many libraries using Gradle when wanting to migrate to Java 9. Many libraries rely on running unit tests and over 28.000 artifacts on the Maven Central repository[15] use SLF4J.

Additionally to this bug, Gradle has another issue, that does not only affect Java 9 or modularization with JPMS, but especially Gradle's multi-module builds in general: Gradle has no central place to manage dependencies of multiple modules across a build. In contrast Maven, has the concept of a *Bill of Materials* (BOM), that lists all dependencies and their versions in the root project. Gradle manages dependencies separately in each build script for every module. This requires to keep track of modules that are shared between different modules and causes additional effort to keep the versions of those dependencies synchronized across every module. Different versions of the dependencies would likely cause unexpected behavior as described in Section 2.2.1.

To work around this problem a custom solution was implemented, that groups dependencies into groups and makes them reusable across different modules[16]. Listing 13 shows an excerpt from the centralized dependency management in Gradle's `buildSrc` project, that is put on the classpath of every build script. In the build scripts the groups of dependencies can then be used as `libraries.libreOffice` without requiring to specify the version locally.

Listing 13 Centralized Dependency Management in Gradle

```
package org.jabref.build
class Dependencies {
    static def libraries = [
            libreOffice: [
                    "org.libreoffice:juh:5.4.2",
                    "org.libreoffice:jurt:5.4.2",
                    "org.libreoffice:ridl:5.4.2",
                    "org.libreoffice:unoil:5.4.2",
            ],
            // >1.8.0-beta is required for java 9 compatibility
            loggingApi: "org.slf4j:slf4j-api:1.8.0-beta2",

            // [...]
    ]
}
```

In contrast, a tool used for the modularization of JabRef supporting Java 9 well was the IDE (Integrated Development Environment) *JetBrains IntelliJ IDEA 2018.1 Ultimate*. For example, when moving parts of an application to a new module, this IDE automatically analyzes the required dependencies of the moved parts and warns developers when some dependencies will not be available in the new module.

In summary the missing support of the used tools for Java 9 modules was the biggest problem for modularizing JabRef.

[15] `https://mvnrepository.com/artifact/org.slf4j/slf4j-api`
[16] `https://github.com/JabRef/jabref/pull/4163`

5 Future Work

In the process of migrating JabRef multiple code contributions were provided to various open source projects including JabRef itself. While a functional version of JabRef could be produced, some parts of the application had to be disabled due to some external libraries not providing updates, that are compatible with Java 9. Due to those issues, the version is not yet released in binary form.

For the future the primary goal should be to get this version released. It is unclear, if the maintainers of those libraries will provide an update in the future, but the parts disabled in JabRef will have to be reenabled before a Java 9 compatible version can be officially released. If the maintainers stay unresponsive, libraries providing similar functionality and being compatible with Java 9 should be adopted. This will require future effort, as it is likely that the APIs of such libraries will differ. In the case that no alternatives are available, the maintenance of the libraries could be taken over, but this would have exceeded the scope of this thesis.

In addition to the compatible version with Java 9, the application was modularized with JPMS. The current modularization resembles the basic components of the high-level architecture (see Section 2.3).

In a future version the modularization could be done much more fine-grained. Especially features of JabRef such as the fetchers, that can fetch bibliography entries form various sources, could be implemented using JPMS' services. But also features outside of JabRef's core functionality, that are not necessarily required and are not used by every user, such as synchronizing bibliography databases with SQL (Structured Query Language) databases or automatically generating bibliographies and citations in third-party applications, could be extracted to additional modules. The services of those modules could be loaded as available, so that they are not required to run JabRef. This would change the architecture of JabRef to a more service oriented architecture (SOA). Also this would reduce maintenance effort of JabRef, as the size of the core code base would shrink, and such extensions could be maintained separately.

Another important point that should be addressed in the future is the support of build tools for Java 9. All of the used tools described in Section 4.2 are publicly available as open- source projects.

6 Conclusion

The Java programming language has experienced high popularity in many enterprises and also from private developers since its initial release in 1996. Since then the Java platform has continuously undergone changes and many features were added. With the latest major addition – the Java Platform Module System (JPMS) – many Java developers have to cope with breaking changes in the platform.

In the first part of this thesis, the challenges when migrating to JPMS were analyzed. First the open source application JabRef was migrated to Java 9. While there were also some problems with the access of internal APIs in the Java Development Kit (JDK), the main problem when upgrading to Java 9 is the unavailability of updates for compatible third-party libraries.

Many very popular and widely used libraries such as *Google Guava* are not yet fully compatible with Java 9. The maintainers of the libraries are often already aware of the incompatibilities or are responsive when they are pointed out to them. A huge problem, however, is the unresponsiveness of maintainers of legacy libraries. While solutions for those problems such as manually patching libraries was proposed, no completely satisfactory solution could be implemented in this thesis.

The problems caused by the modularization of the JDK – internal APIs becoming unavailable – can easily be circumvented by the wide range of utilities provided by the maintainers of the Java programming language. Solving such problems in the long term, however, require developers to migrate to new supported APIs.

In the second part of this thesis, JabRef was decomposed into multiple modules. The main problem encountered was the lacking support of the used tools. Popular and widely used build tools such as *Gradle* still have difficulties supporting Java 9 builds after almost a year since its release. A well-organized architecture is essential for modularizing an application. Cleanly encapsulated components with clear dependencies make modularization basically trivial.

Despite all the encountered problems, such a huge change as the introduction of JPMS is unknown to previous releases of Java. It is questionable, if the accelerated release cycle of Java helped slowing down the adoption of Java 9 in popular libraries and tools, as Java 9 is already outdated and unsupported since March 2018, or if developers will wait until the release of the next LTS release – Java 11 – in September 2018, which will be supported until September 2023 [32].

In conclusion, the introduction of modularization in Java at a language level was a necessary and long awaited change. It is possible that the developers in the Java ecosystem simply are not used to the faster release cycle and will adopt JPMS in the future.

References

[1] Terese Besker, Antonio Martini, and Jan Bosch. "The Pricey Bill of Technical Debt: When and by Whom will it be Paid?" In: *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 2017, pp. 13–23. DOI: 10.1109/ICSME.2017.42.

[2] Aline Brito et al. "Why and how Java developers break APIs". In: *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. 2018, pp. 255–265. DOI: 10.1109/SANER.2018.8330214.

[3] Ned Chapin et al. "Types of software evolution and software maintenance". In: *Journal of Software Maintenance* 13.1 (2001), pp. 3–30. DOI: 10.1002/smr.220.

[4] Iris Clark and Mark Reinhold. *Java SE 9 (JSR 379)*. Ed. by Oracle Corp. 2017. URL: http://cr.openjdk.java.net/~iris/se/9/java-se-9-pr-spec/.

[5] Danny Dig and Ralph E. Johnson. "How do APIs evolve? A story of refactoring". In: *Journal of Software Maintenance* 18.2 (2006), pp. 83–107. DOI: 10.1002/smr.328.

[6] Richard S Hall et al. "OSGi in action". In: *Creating Modular Applications in Java* (2011).

[7] Michael Inden. *Java 9 - Die Neuerungen*. 2018.

[8] JabRef Developers. *High Level Documentation*. 2017. URL: https://github.com/JabRef/jabref/wiki/High-Level-Documentation.

[9] JabRef Developers. *Result analysis of JabRef survey*. 2015. URL: http://www.jabref.org/surveys/2015/analysis.

[10] Koushik Kothagal. *Modular Programming in Java 9*. Aug. 1, 2017.

[11] Sander Mac and Paul Bakker. *Java 9 Modularity*. Sept. 5, 2017.

[12] Andrew J. Malton. "The software migration barbell". In: *ASERC Workshop on Software Architecture*. 2001.

[13] D. Mancl. "Refactoring for software migration". In: *IEEE Communications Magazine* 39.10 (2001), pp. 88–93. DOI: 10.1109/35.956119.

[14] Johannes Martin and Hausi A. Müller. "C to Java Migration Experiences". In: *6th European Conference on Software Maintenance and Reengineering (CSMR 2002), 11-13 March 2002, Budapest, Hungary, Proceedings*. IEEE Computer Society, 2002, pp. 143–153. DOI: 10.1109/CSMR.2002.995799.

[15] Dustin Marx. *Observations From A History of Java Backwards Incompatibility*. Ed. by Inspired by Actual Events. June 6, 2016. URL: https://marxsoftware.blogspot.de/2016/06/java-backwards-incompatibility.html.

[16] Anneliese von Mayrhauser and A. Marie Vans. "Program Comprehension During Software Maintenance and Evolution". In: *IEEE Computer* 28.8 (1995), pp. 44–55. DOI: 10.1109/2.402076.

[17] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. "The evolution of Java build systems". In: *Empirical Software Engineering* 17.4-5 (2012), pp. 578–608. DOI: 10.1007/s10664-011-9169-5.

[18] Sven Meier. *Spin*. Apr. 8, 2007. URL: http://spin.sourceforge.net/solution.html.

[19] Benjamin Muschko. *Gradle in action*. Manning, 2014.

[20] Oracle Corp. *Compatibility*. July 21, 2018. URL: http://www.oracle.com/technetwork/java/javase/compatibility-137541.html.

[21] Oracle Corp. *Compatibility Guide for JDK 8*. July 21, 2018. URL: http://www.oracle.com/technetwork/java/javase/8-compatibility-guide-2156366.html.

[22] Oracle Corp. *Incompatibilities in J2SE 5.0 (since 1.4.2)*. July 21, 2018. URL: http://www.oracle.com/technetwork/java/javase/compatibility-137462.html.

[23] Oracle Corp. *Java 2 Platform v1.3 Compatibility with Previous Releases*. July 21, 2018. URL: http://www.oracle.com/technetwork/java/javase/compatibility-135119.html.

[24] Oracle Corp. *Java 2 Platform, Standard Edition Version 1.4.0 Compatibility with Previous Releases*. July 21, 2018. URL: http://www.oracle.com/technetwork/java/javase/compatibility-j2se1-141394.html.

[25] Oracle Corp. *Java 2 SDK Compatibility with Previous Releases*. July 21, 2018. URL: http://www.oracle.com/technetwork/java/javase/compatibility-141553.html.

[26] Oracle Corp. *Java Platform, Standard Edition Oracle JDK 9 Migration Guide*. 2017. URL: https://docs.oracle.com/javase/9/migrate/toc.htm.

[27] Oracle Corp. *Java Platform, Standard Edition Oracle JDK 9 Migration Guide*. July 21, 2018. URL: https://docs.oracle.com/javase/9/migrate/toc.htm.

[28] Oracle Corp. *Java SE 7 and JDK 7 Compatibility*. July 21, 2018. URL: http://www.oracle.com/technetwork/java/javase/compatibility-417013.html.

[29] Oracle Corp. *Java Timeline*. 2015. URL: http://oracle.com.edgesuite.net/timeline/java/.

[30] Nicolai Parlog. *Java 9 Migration Guide: The Seven Most Common Challenges*. Ed. by CodeFX. July 24, 2017. URL: https://blog.codefx.org/java/java-9-migration-guide/.

[31] William Pugh. *JSR 305: Annotations for Software Defect Detection*. 2006. URL: https://jcp.org/en/jsr/detail?id=305.

[32] Mark Reinhold. "Java in 2018: Change is the Only Constant". In: Devoxx UK 2018. Presented at Devoxx UK 2018. May 10, 2018. URL: https://www.youtube.com/watch?v=HqxZFoY_snQ.

[33] Mark Reinhold. *The State of the Module System*. Ed. by Oracle Corp. Aug. 3, 2016. URL: http://openjdk.java.net/projects/jigsaw/spec/sotms/.

[34] Mark Reinhold and Oracle Corp. *JSR 376: JavaTM Platform Module System*. 2017. URL: https://www.jcp.org/en/jsr/detail?id=376.

[35] Laerte Xavier et al. "Historical and impact analysis of API breaking changes: A large-scale study". In: *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 2017, pp. 138–147. DOI: 10.1109/SANER.2017.7884616.

Ich erkläre hiermit gemäß §17 Abs. 2 APO, dass ich die vorstehende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bamberg, den August 5, 2018 Florian Beetz