UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
Detection and Repair of Architectural Inconsistencies in Java

Permalink
https://escholarship.org/uc/item/4f0711jd

Author
Ghorbani, Negar

Publication Date
2019

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Detection and Repair of Architectural Inconsistencies in Java

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Software Engineering

by

Negar Ghorbani

Thesis Committee:
Professor Sam Malek, Chair
Assistant Professor Joshua Garcia
Assistant Professor Iftekhar Ahmed

2019

© 2019 Negar Ghorbani

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

ACKNOWLEDGMENTS **vi**

ABSTRACT OF THE THESIS **vii**

1 Introduction **1**

2 Java Platform Module System **4**
2.1 JPMS Goals and Potential Misuse 4
2.2 Understanding Java 9 JPMS Modules 6

3 Inconsistent Module Dependencies **11**

4 Darcy **17**
4.1 Detection 18
4.2 Repair 21

5 Evaluation **24**
5.1 RQ1: Pervasiveness 26
5.2 RQ2: Correctness 27
5.3 RQ3: Security 28
5.4 RQ4: Encapsulation 29
5.5 RQ5: Software Bloat 30
5.6 RQ6: Performance 31

6 Threats to Validity **34**

7 Related Work **36**

8 Conclusion **39**

Bibliography **40**

Appendix A Artifacts **46**

LIST OF FIGURES

Page

2.1 Three example modules with their inter-dependencies 7

4.1 A high-level overview of DARCY 17
4.2 Reflective method invocation example 19
4.3 Service loader example 21

LIST OF TABLES

Page

3.1 Functions describing dependencies based on module directives of JPMS . . . 12

5.1 Identified Inconsistencies and Robustness Results 25
5.2 Result for Attack-Surface Reduction 29
5.3 Results for Encapsulation Improvement 32
5.4 Results for Software-Bloat Reduction 33
5.5 Results for Execution Time 33

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my advisors, Professor Sam Malek and Professor Joshua Garcia, who have helped and supported me throughout this project and my research experience with their insightful feedbacks and guidelines.

I would like to thank my committee members, Professor Crista Lopes, Professor Iftekhar Ahmed, and Professor Brian Demsky for reviewing my thesis and providing valuable feedbacks.

This work was supported in part by awards CCF-1252644, CNS-1629771, CCF-1618132, and CNS-1823262 from the National Science Foundation.

ABSTRACT OF THE THESIS

Detection and Repair of Architectural Inconsistencies in Java

By

Negar Ghorbani

Master of Science in Software Engineering

University of California, Irvine, 2019

Professor Sam Malek, Chair

Java is one of the most widely used programming languages. However, the absence of explicit support for architectural constructs, such as software components, in the programming language itself has prevented software developers from achieving the many benefits that come with architecture-based development. To address this issue, Java 9 has introduced the Java Platform Module System (JPMS), resulting in the first instance of encapsulation of modules with rich software architectural interfaces added to a mainstream programming language. The primary goal of JPMS is to construct and maintain large applications efficiently—as well as improve the encapsulation, security, and maintainability of Java applications in general and the JDK itself. A challenge, however, is that module declarations do not necessarily reflect actual usage of modules in an application, allowing developers to mistakenly specify inconsistent dependencies among the modules. In this thesis, we formally define 8 inconsistent modular dependencies that may arise in Java-9 applications. We also present Darcy, an approach that leverages these definitions and static program analyses to automatically (1) detect the specified inconsistent dependencies within Java applications and (2) repair those identified inconsistencies. The results of our experiments, conducted over 38 open-source Java-9 applications, indicate that architectural inconsistencies are widespread and demonstrate the benefits of Darcy in automated detection and repair of these inconsistencies.

Chapter 1

Introduction

A software system's architecture comprises the principal design decisions employed in the system's construction [64]. Although every system has an architecture, the architecture of many systems is not explicitly documented, for instance in the form of UML models. Ensuring that the architecture as documented or intended, known as the *prescriptive* architecture, matches the architecture reflected in the system's implementation, known as the *descriptive* architecture, remains a major challenge [64]. Architecture of a system is often conceptualized in terms of high-level constructs, such as software components, connectors, and their interfaces, while programming languages provide low-level constructs, such as classes, methods, and variables, making it a non-trivial task to map one to the other.

Inconsistencies between prescriptive and descriptive architectures are of utmost concern in any software project, since architecture is the primary determinant of a software system's key properties. One promising approach for abating the occurrence of architectural inconsistencies is to make it easier to bridge the gap between architectural abstractions and their implementation counterparts. To that end, the software-engineering research community has previously advocated for *architecture-based development*, whereby a programming language

(e.g., ArchJava [12]) or a framework (e.g., C2 [65]) provides the implementation constructs for realizing the architectural abstractions.

In spite of this prior work in the academic community, until recently, Java—arguably the most popular programming languages over the past two decades—lacked extensive support for architecture-based development. This all changed with the introduction of Java Platform Module Systems (JPMS) in Java 9. Modules are intended to make it easier for developers to construct large applications, and improve the encapsulation, security, and maintainability of Java applications in general as well as the JDK itself [3].

Using Java's module system, the developer explicitly specifies the system's components (i.e., modules in Java) as well as the specific nature of their dependencies in a file called `module-info`. However, Java 9 does not provide any mechanism to ensure the prescriptive architecture specified in the `module-info` file is in fact consistent with the descriptive architecture of the implemented software, i.e., whether the declared dependencies in the `module-info` file are accurately reflecting the implemented dependencies among the system's components. Inconsistencies between the prescriptive and descriptive architectures in Java 9 matter. The Java platform uses the `module-info` file to determine the level of access granted to each module, and to determine which modules should be packaged together for deployment. As a result, inconsistencies between prescriptive and descriptive architecture in Java have severe security and performance consequences. These inconsistencies also affect the engineers ability to use the prescriptive architecture for understanding the system's properties or to make maintenance decisions.

In this thesis, we formally define 8 modular inconsistencies that may occur in Java-9 applications. We present DARCY, an approach that leverages these definitions and static analyses to automatically (1) detect the specified inconsistencies within Java applications and (2) repair them. DARCY is also publicly available [10].

The results of our experiments, conducted over 38 open-source Java-9 applications, indicate that architectural inconsistencies are widespread, and demonstrate the benefits of DARCY in automated detection and repair of these inconsistencies. DARCY found 124 instances of inconsistencies among 38 Java applications in our data set. By automatically fixing these inconsistencies, DARCY was able to measurably improve various attributes of the subject applications' architectures by reducing the attack surface of applications by 60.33%, improving their encapsulation by 23.03%, and producing deployable applications that consume 14.02% less memory.

The remainder of this thesis is organized as follows. Section 2 introduces the module system of Java 9 and its design goals. Section 3 formally specifies the architectural inconsistencies in the context of Java 9. Section 4 provides details of our approach and its implementation. Section 5 presents the experimental evaluation of the research. Section 6 includes the threats to validity of our approach. The thesis concludes with an outline of related research and future work.

Chapter 2

Java Platform Module System

To aid the reader with understanding architectural specification in Java 9, we introduce the new module system for Java 9, called Java Platform Module System (JPMS). We overview JPMS's goals and the architectural risks that arise from its misuse. We then discuss the details of modules in Java 9—including module declarations and module directives.

2.1 JPMS Goals and Potential Misuse

JPMS enables specification of a prescriptive architecture in terms of key architectural elements—specifically components in the form of Java-9 modules, architectural interfaces, and resulting dependencies among components. JPMS aims to enable reliable configuration, stronger encapsulation, modularity of the Java Development Kit (JDK) and Java Runtime Environment (JRE) to solve the problems faced by engineers when developing and deploying Java applications [60].

Software designers and developers can achieve strong encapsulation in their Java-9 systems by modularizing them and allowing explicit specification of interfaces and dependencies.

Encapsulation in Java 9 is achieved by allowing architects or developers to specify which of a Java-9 module's public types are accessible or inaccessible to other modules [55]. A module must explicitly declare which of its public types are accessible to other modules. A module cannot access public types in another module unless those modules explicitly make their public types accessible. As a result, JPMS has added more refined accessibility control—allowing architects and developers to decrease accessibility to packages, reduce the points at which a Java application may be susceptible to security attacks, and design more elegant and logical architectures [22].

Prior to Java 9, the Java platform was a monolith consisting of a massive number of packages, making it challenging to develop, maintain, and evolve. Software developers could not easily choose a subset of the JDK as a platform for their applications. This results in software bloat and more potential points of attack for malicious agents. With the introduction of JPMS in Java 9, the Java platform is now modularized into 95 modules. Furthermore, many internal APIs are hidden from apps using the platform [55], potentially reducing problems involving software bloat and security.

Using JPMS in Java 9, Java developers can now create lightweight custom JREs consisting of only modules they need for their application or the devices they are targeting. As a result, the Java platform can more easily scale down to small devices, which is important for microservices or IoT devices [23]. For example, if a device does not support GUIs, developers could use JPMS to create a runtime environment that does not include the GUI modules, significantly reducing the runtime memory size[22].

Although JPMS allows for specification of prescriptive architectures, the descriptive architecture of a Java application may be inconsistent with the prescriptive architecture. Such inconsistencies may arise due to architects or developers misunderstanding of a software systems' architectures (e.g., an architect mistakenly specifies a more accessible interface than he intended), or simply due to mistaken implementations (e.g., a developer neglects to use

a module's interface, even though the architect intended such a use). This can result in (1) a poorly encapsulated architecture, making an application harder to understand and maintain; (2) bloated software; or (3) insecure software. In terms of security, for instance, one of the potential problems is the granting of unnecessary access to internal classes and packages, potentially resulting in security vulnerabilities. In terms of software bloat, inconsistent dependencies can compromise scalability and performance of Java software (e.g., requiring many unnecessary modules from the JDK).

2.2 Understanding Java 9 JPMS Modules

In JPMS, a *module* is a uniquely named, reusable group of related packages, as well as resources (such as images and XML files)[3]. Each module has a descriptor file, `module-info.java`, which contains meta-data, including the declaration of a named module. A named module should specify (1) its dependencies on other modules, i.e., the classes and interfaces that the module needs or expects, and should specify (2) which of its own packages, classes, and interfaces are exposed to other modules.

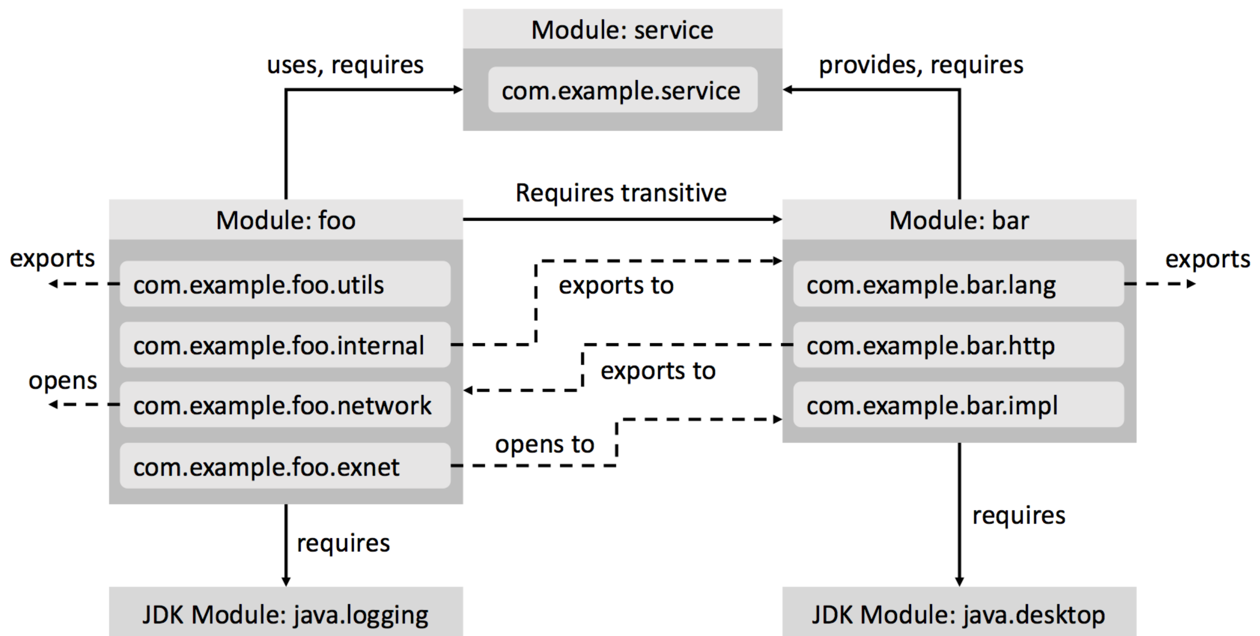
A module can be a *normal module* or an *open module*. A normal module allows access from other modules at compile time and run time to only explicitly exported packages; an open module allows access from other modules (1) at compile time to only explicitly exported packages and (2) at run time to all its packages [30].

The module declaration file consists of a unique module name and a module body. Any module body can be empty or contain one or more module directives, which specifies a module's exposure to other modules or the modules it needs access to.

Figure 2.1 shows an example of a project with three modules: `bar`, `foo`, and `service`. The declarations of each module provided in its `module-info.java` file is described in Figure

```
1  module bar {
2      requires java.desktop;
3      requires service;
4
5      exports com.example.bar.lang;
6      exports com.example.bar.http to foo;
7
8      provides com.example.service.Srv with com.example.
           bar.impl.ImplService; }
9
10 module foo {
11     requires service;
12     requires java.logging;
13     requires transitive bar;
14
15     exports com.example.foo.utils;
16     exports com.example.foo.internal to bar;
17
18     opens com.example.foo.network;
19     opens com.example.foo.exnet to bar;
20
21     uses com.example.service.Srv;}
22
23 module service {
24     exports com.example.service;}
```

(a) Module declarations and their directives provided in their *module-info.java* files.



(b) Specified dependencies between modules based on their directives

Figure 2.1: Three example modules with their inter-dependencies

2.1a. Figure 2.1b is a diagram that depicts the relationship between the same modules based on dependencies in their declarations.

A module body can utilize combinations of the following five module directives [30], which specify module interfaces and their usage: the *requires* directive specifies the packages that a module needs access to, the *exports* and *opens* directives make packages of a module available to other modules, the *provides* directive specifies the services a module provides, and the *uses* directive specifies the services a package consumes. These directives can be declared as described below:

- The `requires` directive with declaration `requires` m_2 of a module $m1$ specifies the name of a module m_2 that m_1 depends on. m_2 can be a user-defined module or a module within the JDK. For example, in Figure 2.1, module `bar` requires module `java.desktop`. The requires declaration of a module m_1 may be followed by the `transitive` modifier, which ensures that any module m_3 that requires m_1 also implicitly requires module m_2. As an example, in Figure 2.1, module `foo` requires module `bar` and any module that requires `foo` also implicitly requires `bar`.

- The `exports` directive with declaration `exports` p of a module m_1 specifies that m_1 exposes package p's public and protected types, and their nested public and protected types, to all other modules at both runtime and compile time. For example, in Figure 2.1, the module `bar` exports the package `com.example.bar.lang`. We can also export a package specifically to one or more modules by using the `exports` p `to` $m_2, m_3, ..., m_n$ declaration. In this case, the public and protected types of the exported package are only accessible to the modules specified in the *to* clause.

  As an example, in Figure 2.1, module `foo` exports `com.example.foo.internal` to the module `bar`.

- The `opens` directive with declaration `opens` p specifies that package p's nested pub-

lic and protected types, and the public and protected members of those types, are accessible by other modules at runtime but not compile time. This directive also grants reflective access to all types in p, including the private types, and all its members, from other modules. For example, in Figure 2.1, module `foo` makes package `com.example.foo.network` available to other modules only at runtime, including through reflection. This directive may also be followed by the *to* modifier, resulting in the `opens` p `to` $m_2, m_3, ..., m_n$ declaration. In this case, the public and protected types of p are only accessible to the modules specified in the *to* clause. For instance, in Figure 2.1, module `foo` makes package `com.example.foo.exnet` available only at runtime, including through reflection, to the module `bar`. Unlike the other directives that can only be used in the body of a module's specification, `open` can be used in both the body of a module's specification and in its header (i.e., before the module's name). The latter usage is a shorthand way of denoting all packages in the module are `open`.

- The `provides with` directive with declaration `provides` c_1 `with` $c_2, c_3, ..., c_n$ of module m_1 specifies that a class c_1 is an abstract class or interface that is provided as a service by m_1. The *with* clause specifies one or more service provider classes for use with `java.util.ServiceLoader`. A service is a well-known set of interfaces and (usually abstract) classes. A service provider is a specific implementation of a service. `java.util.ServiceLoader<S>` is a simple service-provider loading facility. It loads a provider implementing the service type `S` [1]. For instance in Figure 2.1, module `bar` provides the abstract class `com.example.service.Srv` as a service using the `com.example.bar.impl.ImplService` class as the service's implementation.

- The `uses` directive with declaration `uses` c_1 of a module m_1 specifies that m_1 uses a service object of an abstract class or interface, c_1, provided by another module. For this purpose, the module should discover providers of the specified service via

`java.util.ServiceLoader`. As an example from Figure 2.1, module `foo` uses the service object of class `com.example.service.Srv`, which is provided by module `bar`.

Note that, as depicted in Figure 2.1, both `provides with` and `uses` directives need the module being declared to `require` the service module as well.

Chapter 3

Inconsistent Module Dependencies

Based on the module directives described in the previous section, inconsistencies may arise when using modules. Insufficiently specified dependencies (e.g., a module that attempts to use a package it does not have a requires directive for) are already checked by the Java platform. However, *excess dependencies*, where a module either (1) exposes more of its internals than are used or (2) requires internals of other modules that it never uses, are not handled by Java. These inconsistencies can affect various architectural attributes:

A1: Encapsulation and Maintenance—Requiring unneeded functionalities of other modules increases the complexity of the module unnecessarily, compromises its encapsulation, and decreases its maintainability.

A2: Software Bloat and Scalability—Requiring unneeded modules, especially from JDK, can result in bloated software, which compromises scalability of the application.

A3: Security—Excessively exposing the internals of a module can result in errors or security issues arising in the module.

To achieve a systematic and comprehensive coverage of all types of inconsistent module

Table 3.1: Functions describing dependencies based on module directives of JPMS

Function	**Description**
$Req(m_1, m_2)$	Module m_1 requires module m_2
$ReqJDK(m_1, m_{jdk})$	Module m_1 requires the JDK module m_{jdk}
$ReqTransitive(m_1, m_2)$	Module m_1 requires transitive module m_2
$Exp(m, p)$	Module m exports package p
$ExpTo(m_1, p_1, \{m_2, m_3, \ldots\})$	Module m_1 exports package p_1 to the set of modules $\{m_2, m_3, \ldots\}$
$Open(m)$	Module m is open
$Opens(m, p)$	Module m opens package p
$OpensTo(m_1, p, \{m_2, m_3, \ldots\})$	Module m_1 opens package p to the set of modules $\{m_2, m_3, \ldots\}$
$Uses(m, s)$	Module m uses Service s
$ProvidesWith(m, s, \{c_1, c_2, \ldots\})$	Module m provides service s with the set of classes $\{c_1, c_2, \ldots\}$
$LoadsService(c, s)$	Class c loads Service s via the `java.util.ServiceLoader` API
$Dep(p_1, p_2)$	Source code in package p_1 uses classes of package p_2
$ReflDep(p_1, p_2)$	Source codes in package p_1 uses classes of package p_2 via reflection

dependencies, we studied all potential inconsistencies resulting from developers' misuse of each type of module directive. In the remainder of this section, we focus on specifying eight types of inconsistent dependencies that may arise when using JPMS and the functions needed to specify those dependencies.

Table 3.1 includes 11 functions that directly model different variations of the five module directives in JPMS. To describe a class loading a service using `java.util.ServiceLoader` API, we define the *LoadsService* function. For actual code usage among packages, as opposed to those specified through module directives, we define the *Dep* function.

By leveraging the functions in Table 3.1, we introduce eight types of excess inconsistent dependencies: requires, JDK requires, requires transitive, exports(to), provides with, uses, open, and opens(to) modifiers. For each inconsistent dependency type, there is a dependency explicitly defined in a `module-info` file which is not actually used in the source code of the module. Using these formal definitions, Section 4 detects and repairs the following inconsistent dependencies.

Inconsistent Requires Dependency: This scenario describes an inconsistent *requires* dependency in which (1) module m_1 explicitly declares that it requires another module m_2 and (2) no class of m_1 actually uses any class inside exported packages of m_2. As a result, this inconsistency mostly affects attribute A1. It can also affect attribute A2.

$$Req(m_1, m_2) \land (\nexists\ p_1 \in m_1,\ p_2 \in m_2 : Dep\ (p_1, p_2)) \tag{3.1}$$

Inconsistent JDK Requires Dependency: This scenario describes an inconsistent *requires* dependency in which module m_1 explicitly declares that it requires a module inside the Java JDK, m_{jdk}. However, none of the classes inside m_1 uses any class inside exported packages of m_{jdk}. Hence, it affects attribute A1, and more importantly A2. We distinguish this scenario from the previous one because an inconsistency involving JDK modules has a greater effect on portability than the previous more generic scenario.

$$Req(m_1, m_{jdk}) \land (\nexists p_1 \in m_1, p_2 \in m_{jdk} : Dep(p_1, p_2)) \tag{3.2}$$

Inconsistent Requires Transitive Dependency: An excess *transitive* modifier in a *requires* dependency consists of the following (1) a module m_1 explicitly declares in its `module-info` file that it transitively requires another module m_2—which means any module that requires m_1 also implicitly requires m_2; and (2) no class of a module that requires $m1$ actually uses any class in $m2$. This type of inconsistency mostly affects attribute A1, but also affects A2.

$$\begin{gathered} ReqTransitive(m_1, m_2) \land (\forall\ m : Req(m, m_1), \\ \forall\ p \in m, \forall\ p_2 \in m_2 : \neg Dep(p, p_2)) \end{gathered} \tag{3.3}$$

Inconsistent Exports/Exports to Dependency: An inconsistent *exports* dependency occurs when a module m_1 explicitly exports a package p_1 to all other modules, while no package in those other modules use p_1.

$$Exp(m_1, p_1) \land (\forall\ p \notin m_1 : \neg Dep(p, p_1)) \tag{3.4}$$

For an *exports to* directive, this inconsistency occurs when m_1 exports the package p_1 to a specific list of modules M, while no class outside m_1, or inside module list M, uses any class inside p_1.

$$ExpTo(m_1, p_1, M) \land (\forall\ p \in M : \neg Dep(p, p_1)) \tag{3.5}$$

These inconsistencies mostly affect attribute A3 by granting unnecessary access to classes and packages. They also affect attribute A1 due to complicating the architecture.

Inconsistent Provides With Dependency: An inconsistent *provides with* dependency has two key parts: (1) a module m explicitly declares that it provides a service s, which is an abstract class or interface that is extended or implemented by a set of classes $E = \{c_1, c_2, ..., c_k\}$ inside m; and (2) none of the classes inside other modules uses service s via the `java.util.ServiceLoader` API. Consequently, this inconsistency type—similar to inconsistent `requires` dependency—affects attribute A1 and A2 because the `provides with` dependency necessitates a `requires` directive as well. Additionally, this inconsistency type grants unnecessary access to a subset of the application's classes via the `ServiceLoader` API which affects attribute A3.

$$ProvidesWith(m, s, E) \land (\forall m' \neq m : \neg Uses(m', s)) \tag{3.6}$$

Inconsistent Uses Dependency: An inconsistent *uses* dependency occurs when (1) a module m explicitly declares in its `module-info.java` file that it uses a service s and (2)

none of the classes inside m actually use the service s via the `java.util.ServiceLoader` API. This inconsistency type, similar to the previous type, will affect attribute A1 and A2, due to adding an additional `requires` directive.

$$Uses(m, s) \wedge (\forall\ c \in m : \neg\ LoadsService(c, s)) \tag{3.7}$$

Inconsistent Open Modifier: An excess *open* modifier occurs in the following scenario: (1) a module m declares that it opens all its packages to all other modules—recall from Section 2.2 that unlike the other directives, `open` can be used in the header of a module's specification to denote all its packages are open; and (2) there is at least one package p inside m that no class outside m reflectively accesses. As a result, any such package p is potentially open to misuse through reflection, e.g., external access to private members of a class that should not be allowed by any other class. This inconsistency type will affect attribute A3—and make the architecture inaccurate and more complicated, affecting attribute A1.

$$Open(m) \wedge (\exists p \in m : \forall p' \notin m : \neg ReflDep(p', p) \tag{3.8}$$

Inconsistent Opens/Opens To: An inconsistent *opens* dependency occurs when a module m declares that it opens a package p to all other modules via reflection, while none of the classes outside m reflectively accesses any classes of package p.

$$Opens\ (m, p) \wedge \forall\ p' \notin m : \neg ReflDep(p', p) \tag{3.9}$$

Similarly, for *opens to*, the *to* modifier specifies a list of modules M for which module m opens a package p to access via reflection, while no package of m reflectively accesses p.

$$OpensTo(m, p, M) \wedge \forall\ p' \in M : \neg ReflDep(p', p) \tag{3.10}$$

For these inconsistency types, private members of p are open to dangerous misuse through undesired access and reflection, affecting attribute A3, and can also affect attribute A1 due to unnecessarily complicating the architecture.

Chapter 4

Darcy

In the previous section, we introduced various types of inconsistent dependencies. This section describes how we leverage these definitions to design and implement Darcy. Figure 4.1 depicts a high-level overview of Darcy comprised of two phases, *Detection* and *Repair*. Darcy is implemented in Java and Python.

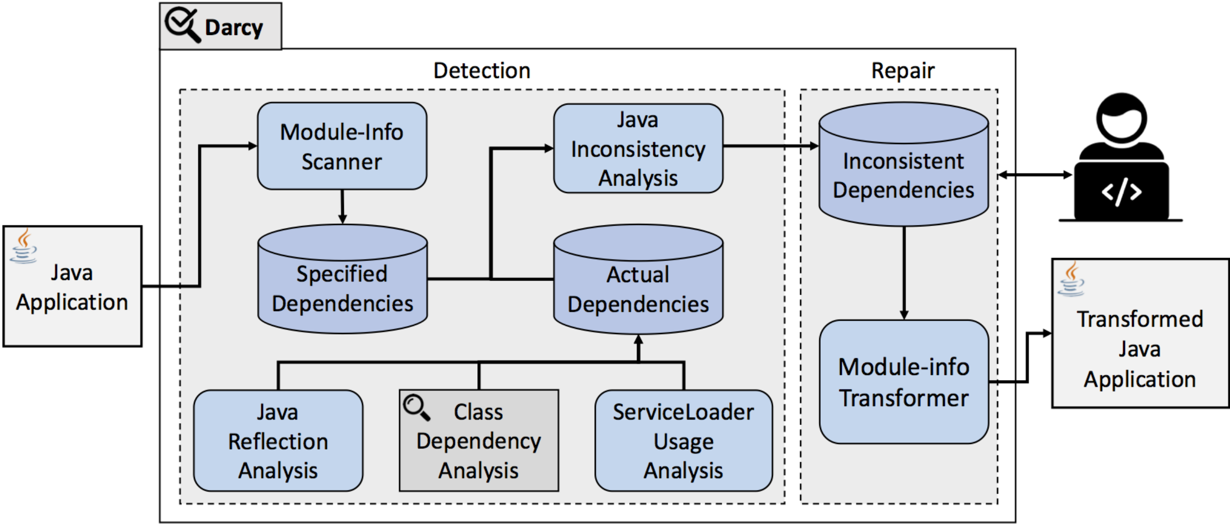


Figure 4.1: A high-level overview of Darcy

4.1 Detection

The detection phase takes a Java application as input and identifies any instance of the eight inconsistent dependencies described in Section 3.

To identify actual dependencies of an input Java application, DARCY relies on static analysis, represented as *Class Dependency Analysis* in Figure 4.1. In the implementation of DARCY, we leveraged Classycle [26] for *Class Dependency Analysis*. More precisely, the information about actual dependencies in the source code of the input application is collected by running Classycle, which provides a complete report of all dependencies in source code of a Java application at both the class and package levels. We only need the extracted dependencies among packages since the dependencies defined in modules are at the package level. *Class Dependency Analysis's* results are stored in *Actual Dependencies*, which is a database component.

A Java application may contain multiple modules, each with a `module-info` file describing the module's dependencies. For extracting a prescriptive architecture, we developed *Module-Info Scanner* which examines all `module-info.java` files within the input Java application and extracts all specified dependencies which are defined at the package level. The collected information of specified dependencies are stored in another database component, *Specified Dependencies*.

Java Reflection Analysis leverages a custom static analysis [27], which we have implemented using the Soot framework [68], to identify usage of reflection in the input application. The traces of any actual usage of reflection in the Java application is then stored in *Actual Dependencies*.

Java Reflection Analysis extracts reflective invocations that occur in cases where non-constant strings, or inputs, are used as target methods of a reflective call. Reflective in-

vocation of a method, for both constructor and non-constructor methods, occurs in three stages: (1) class procurement (i.e., a class with the method of interest is obtained) (2) method procurement (i.e., the method of interest to be invoked is identified), and (3) the method of interest is actually invoked. *Java Reflection Analysis* attempts to identify information at each stage.

```
1  ClassLoader cl = MyClass.getClassLoader();
2  try { Class c = cl.loadClass("NetClass");
3        ...
4        Method m = c.getMethod("getAddress",...);
5        ...
6        m.invoke(...); }
7  catch { ... }
```

Figure 4.2: Reflective method invocation example

A simple example, based on those found in real-world apps, of reflective method invocation, not involving constructors, is depicted in Figure 4.2. In this example, a `ClassLoader` for `MyClass` is obtained (line 1), which is responsible for loading classes. The `NetClass` class is loaded using that `ClassLoader` (line 2). The `getAddress` method of `NetClass` (line 4)—which performs network operations—is retrieved and eventually invoked using reflection (line 6).

Our analysis identifies reflectively invoked methods using a backwards analysis. That analysis begins by identifying all reflective invocations (e.g., line 6 in Figure 4.2). Next, the analysis follows the use-def chain of the invoked `java.lang.reflect.Method` instance (e.g., `m` on line 6) to identify all possible definitions of the `Method` instance (e.g., line 4). Our analysis considers various methods that return `Method` instances, i.e., using `getMethod` or `getDeclaredMethod` of `java.lang.Class`. The analysis then records each identified method name. If the analysis cannot resolve the name, this information is also recorded. In this case, the analysis conservatively indicates that any method of the package opened for reflection

can be accessed.

For constant strings, the analysis attempts to identify the class name that is being invoked. Similar to the resolution of method names, the analysis follows the use-def chain of the `java.lang.Class` instance from which a `java.lang.Class` is retrieved (e.g., following the use-def chain of `c` on line 4). We model various means of obtaining a `java.lang.Class` instance. For example, the class may be loaded by name using a `ClassLoader`'s `loadClass(...)` method (e.g., line 2), using `java.lang.Class`'s `forName` method, or through a class constant (e.g., using `NetClass.class`). The analysis then records the class name it can find statically, or stores that it could not resolve that name. Note that our analysis considers any subclass of `ClassLoader`. Our reflection analysis involving constructors works in a similar manner by analyzing invocations of `java.lang.reflect.Constructor` and invocations of its `newInstance` method.

Similar to our analyses for reflectively invoked methods, We perform analyses for any `set*` methods of `java.lang.reflect.Field` (e.g., `setInt(...)`) or `get*Field*` methods of `java.lang.Class` (e.g., `getDeclaredField(String)`).

For extracting the actual dependencies of type *uses* we implemented *ServiceLoader Usage Analysis* which leverages a custom static analysis using the Soot framework to identify usage of `java.util.ServiceLoader` in the input application. The traces of any actual usage of a service is then stored in *Actual Dependencies*.

An application obtains a service loader for a given service by invoking the static `load` method of `ServiceLoader` API. A service loader can locate and instantiate providers of the given service using the `iterator` or `stream` method [6], through which an instance of each of the located service providers can be created. As an example, Figure 4.3 depicts the code that obtains a `ServiceLoader` for `MyService` (line 1). The `ServiceLoader` loads providers of `MyService` (line 2) and can instantiate any of the located providers of this service using its

iterator—created by the for loop in line 3. In this example, the service provider with the `getService` method is desired (line 4).

```
1   ServiceLoader<MyService> loader;
2   loader = ServiceLoader.load(MyService.class);
3   for (MyService s : loader) {
4       if (s.getService != null){... } }
```

Figure 4.3: Service loader example

Our analysis identifies the usage of the `ServiceLoader` API using a backward analysis by following the use-def chain of `ServiceLoader` instances (e.g., `s` on line 4) to identify all possible definitions of a `ServiceLoader` (e.g, line 2 in Figure 4.3). The results of the `ServiceLoader` API usage is then stored in *Actual Dependencies*.

Java Inconsistency Analysis's main goal is to identify all types of inconsistency scenarios described in Section 3. For each directive in a `module-info.java` file, *Java Inconsistency Analysis* explores actual and specified dependencies, stored in their respective database components, to identify any occurrence of an inconsistent dependency defined in Section 3. If a matching instance is found, *Java Inconsistency Analysis* reports the identified architectural inconsistency, the module affected, and the specific directive involved. The component then stores the identified inconsistencies in *Inconsistent Dependencies*, which are then used in the repair phase.

4.2 Repair

To repair inconsistent dependencies, *Module-Info Transformer* deletes or modifies the explicit dependencies defined in the `module-info` files. Inconsistencies found in the previous phase are all unnecessarily defined dependencies among an application's modules and packages.

Therefore, *Module-Info Transformer* needs to omit those inconsistent dependencies specified in the `module-info` files.

The result of the detection phase includes the type and details of identified inconsistencies. For instance, in the case of an inconsistent *exports* dependency, one result stored in *Inconsistent Dependencies* includes the module in which this dependency is specified, the type of the inconsistent dependency (*exports* in this case), and the package that is unnecessarily exported. The repair phase takes the results of the detection phase as input. For each module, the repair phase finds the related records of inconsistent dependencies defined in that module and modifies the affected lines in `module-info`.

For this purpose, we leveraged ANTLR [4] to transform the `module-info.java` files to repair the inconsistent dependencies. ANTLR is a parser generator for reading, processing, executing, or translating a structured text. Hence, we generated a customized parser using Java-9 grammar so that we can modify it to check the records of inconsistent dependencies found in the detection phase of Darcy.

More precisely, we have implemented the generated parser so that, if it finds any match between the tokens of `module-info` files and the inconsistent dependencies, it skips or modifies the specific token with respect to the type of the inconsistency. As a result, depending on the type of dependency, the corresponding line in the `module-info` file is omitted or modified.

Module-Info Transformer repairs each type of inconsistent dependency. In most cases, *Module-Info Transformer* deletes the entire statement. However, for `requires transitive`, *Module-Info Transformer* only removes the token `transitive`.

In case of inconsistencies involving `open` module m (Equation 3.8 in Section 3), the `open` modifier is removed from the header of the module declaration. However, there may be some packages in m that other modules reflectively access. For each of these packages, *Module-Info Transformer* adds an `opens to` statement thats make private members of the package

accessible to the modules that reflectively access the package. If there is no package in m that is reflectively accessed by other modules, no statement will be added to the module's body.

In certain situations, the Darcy user may disagree with the way it repairs and modifies the specified dependencies because Darcy is not aware of the architect's or developer's intentions. For example, this situation may occur if the user wants to develop a library and export some packages for further needs or even allow other modules to reflectively access the internals of some classes and packages. Darcy warns the developers and architects about potential threats caused by architectural inconsistencies in their Java application, and allows them to override Darcy prior to application of repairs.

Chapter 5

Evaluation

To assess the effectiveness of DARCY, we study the following research questions:

RQ1: How pervasive are inconsistent, architectural dependencies in practice?

RQ2: How accurate is DARCY at detecting inconsistent, architectural dependencies and repairing them?

RQ3: To what extent does DARCY reduce the attack surface of Java modules?

RQ4: To what extent does DARCY enhance encapsulation of Java modules?

RQ5: To what extent does DARCY reduce the size of runtime memory?

RQ6: What is DARCY's runtime efficiency in terms of execution time?

To answer these research questions, we selected a set of Java applications from GitHub [5], a large and widely used open-source repository of software projects, all of which are implemented in Java 9. For this purpose, we searched through Java applications in GitHub and selected projects that contain a `module-info.java` file. Our search covered about a hundred pages of search results in the GitHub repository. To assess module dependencies, projects needed to have more than one module in their respective `module-info.java` files. Our final evaluation dataset resulted in 38 Java-9 applications, avoiding any selection bias

Table 5.1: Identified Inconsistencies and Robustness Results

No.	Application Name	# Modules	# Directives	# Total Incons.	Inconsistencies Types							% Correct Incons.	Compiled (After Repair)	Test Passing Rate (%)	
					R	R.J.	R.T	E	P	U	O			Before	After
1	sense-nine	6	31	9	2	2	-	2	-	-	3	100	✓	-	-
2	number-to-text	3	11	1	-	-	-	1	-	-	-	100	✓	100	100
3	vstreamer	6	25	4	1	-	-	-	-	3	-	100	✓	-	-
4	jigsaw-resources	2	5	1	-	-	-	1	-	-	-	100	✓	-	-
5	JavaUtils	6	36	29	-	14	9	6	-	-	-	100	✓	-	-
6	BunnyHop	2	28	17	-	1	2	11	-	-	3	100	✓	-	-
7	java9-modules	2	5	1	-	-	-	1	-	-	-	100	✓	-	-
8	jwtgen	2	13	2	1	-	-	1	-	-	-	100	✓	-	-
9	project-constantin	4	9	5	-	-	1	4	-	-	-	100	✓	100	100
10	java-spi-example	6	26	4	-	-	1	3	-	-	-	100	✓	-	-
11	codersonbeer-app	4	13	4	1	-	-	2	-	-	1	100	✓	-	-
12	rahmnathan-utils	3	14	7	-	1	-	6	-	-	-	100	✓	-	-
13	auto-sort	3	13	1	-	-	-	1	-	-	-	100	✓	97	97
14	java9-demo	4	10	1	-	-	-	1	-	-	-	100	✓	-	-
15	java9-modules-tlb	5	12	1	-	-	-	1	-	-	-	100	✓	-	-
16	java-9-lab	5	15	1	-	-	-	1	-	-	-	100	✓	-	-
17	meetup-16	4	14	6	-	-	-	3	3	-	-	100	✓	-	-
18	java-9-bookstore	6	17	3	-	-	2	-	1	-	-	100	✓	-	-
19	springuni-java9	3	6	3	2	-	-	1	-	-	-	100	✓	-	-
20	java-9-modularity	4	11	1	-	-	1	-	-	-	-	100	✓	-	-
21	java-9-spring-mvn	3	18	8	2	-	-	6	-	-	-	100	✓	-	-
22	music-ui-start	3	15	4	-	-	1	1	-	-	2	100	✓	-	-
23	java9-labs	4	10	4	-	-	-	4	-	-	-	100	✓	-	-
24	practical-security	4	20	2	-	1	-	1	-	-	-	100	✓	-	-
25	java9-junit	3	13	1	-	-	-	1	-	-	-	100	✓	-	-
26	the-message	3	16	1	-	-	-	1	-	-	-	100	✓	-	-
27	jigsaw-tst	4	11	1	-	-	-	1	-	-	-	100	✓	-	-
28	TRPZ	4	19	2	-	-	-	1	-	-	1	100	✓	-	-

toward our approach.

5.1 RQ1: Pervasiveness

Table 5.1 shows, for each application, the total number of inconsistent dependencies Darcy found, modules, module directives used, and inconsistent dependencies by type. 74% of applications in our dataset (28 out of 38) have inconsistent dependencies. Recall that even one existing inconsistent dependency could cause undesired behaviors, or issues with encapsulation, security, or memory utilization (see Section 3).

As depicted in Table 5.1, most of the inconsistent dependencies are of types *exports* or *requires* because these two types of directives are used more frequently than others. The high frequency of inconsistent *exports* dependencies indicates that granting unnecessary access to internal packages are quite common in Java-9 applications, which could cause security vulnerabilities. Among the inconsistent *requires* dependencies, the *requires JDK* dependency occurred more than others, which increases the risk of loading unnecessary JDK modules and compromising portability.

Table 5.1 indicates that a few applications have inconsistent dependencies of type *provides with*, and only one application has an inconsistent *uses* dependency. In fact, these directives are rare compared to other directives. For *provides with* and *uses*, Java checks most of the requirements for avoiding inconsistent dependencies at compile time. Therefore, the possibility of defining an inconsistent *provides with* and *uses* dependencies decreases. Nevertheless, Darcy covers the inconsistent dependencies corresponding to these two directives because they are risky and may appear more frequently in future usage of Java 9.

5.2 RQ2: Correctness

To answer RQ2 for Darcy's detection capability, we ran the detection phase for each of the Java-9 applications in our evaluation dataset to assess whether Darcy can accurately detect inconsistent dependencies. To that end, we manually checked the inconsistent dependencies found by Darcy to ensure their correctness. More precisely, we compared the corresponding record in both *Actual Dependencies* and *Specified Dependencies* to verify the correctness of the inconsistencies discovered by the detection phase. The result, as described in Table 5.1, shows that all inconsistent dependencies found by Darcy are correct.

To evaluate Darcy's ability to correctly repair inconsistencies, we ran the repair phase of Darcy for each of the Java-9 applications in our evaluation dataset to assess whether Darcy repairs the detected inconsistencies without introducing any unexpected behavior. To assess correctness of a repair, we (1) check if each application compiles successfully after running the repair phase and (2) if the application contains a test suite, determine if the application obtains the same *test passing rate*, i.e., the ratio of the number of passing test cases to the total number of test cases, both before and after repairs. We also ran the detection phase after the repair actions. The result showed zero inconsistencies within the transformed Java applications.

The results for compilation after repair are shown in Table 5.1, indicating that all the applications compiled successfully. This confirms that the inconsistent dependencies have been repaired robustly in a way that does not prevent compilation of the applications. Additionally, three applications in our study contain a test suite. The passing rate for each of these test suites remains the same both before and after Darcy repairs, demonstrating that Darcy does not negatively affect expected behavior of repaired applications.

5.3 RQ3: Security

To assess Darcy's ability to enhance security, we consider the *attack surface* of Java-9 applications. The attack surface of a system is the collection of points at which the system's resources are externally visible or accessible to users or external agents. Manadhata et al. introduced an attack-surface metric to measure the security of a system in a systematic manner [44, 45, 46]. Every externally accessible system resource can potentially be part of an attack and, hence, contributes to a system's attack surface. This contribution reflects the likelihood of each resource being used in security attacks. Intuitively, the more actions available to a user or the more resources that are accessible through these actions, the more exposed an application is to security attacks [44, 45, 46].

For a Java-9 application, the main resource under consideration is a Java module. As a result, we define the attack surface of an application as the number of packages that are accessible from outside its modules. To measure the attack surface of Java-9 applications, we count the number of packages exposed by *exports (to)* and *open(s to)* directives. These directives make internals of packages accessible to other modules.

As shown in Table 5.2, 25 out of 29 applications had an average attack-surface reduction of about 60%. Darcy was able to totally eliminate the attack surface in 5 applications.[1] Although eliminating the module-based attack surface does not result in perfect security, Darcy can maximize protection to the asset (i.e., Java packages) through a module's interfaces by eliminating all unnecessary exports and opens directives of the module—other attack vectors (e.g., IPC over network sockets) still remain but are out of scope for Darcy. The relatively large reduction of the attack surface in applications achieved by Darcy indicates that it can significantly curtail security risks in Java-9 applications.

[1] These applications are essentially software utilities or libraries including different modules that provide functionalities for different situations, but do not have any dependency on one another.

Table 5.2: Result for Attack-Surface Reduction

Application Name	# exposed pckg (before)	# exposed pckg (after)	Attack Surface Reduction (%)
sense-nine	6	1	83.33
number-to-text	4	3	25.00
jigsaw-resources	2	1	50.00
JavaUtils	7	1	85.71
BunnyHop	16	2	87.50
java9-modules	2	1	50.00
jwtgen	1	0	100
project-constantin	4	0	100
java-spi-example	6	3	50.00
codersonbeer-app	4	1	75.00
rahmnathan-utils	6	0	100
auto-sort	2	1	50.00
java9-demo	2	1	50.00
java9-modules-tlb	5	4	20.00
java-9-lab	3	2	33.33
meetup-16	4	1	75.00
springuni-java9	1	0	100
java-9-spring-mvn	6	0	100
music-ui-start	5	2	60.00
java9-labs	5	1	80.00
practical-security	4	3	25.00
java9-junit	4	3	25.00
the-message	10	9	10.00
jigsaw-tst	3	2	33.33
TRPZ	5	3	40.00
Avg. Attack Surface Reduction			60.33%

5.4 RQ4: Encapsulation

To evaluate the ability of Darcy to enhance the encapsulation of Java-9 applications, we leveraged two metrics selected from an extensive investigation by Bouwers et al. [15] about the quantification of encapsulation for implemented software architectures. We selected metrics that involve architectural dependencies and are appropriate for the context of modules in JPMS and Java-9 applications.

The first metric we selected is Ratio of Coupling (RoC) [16], which measures coupling among

an application's modules. For Java-9 modules, RoC is the ratio of the *number of existing dependencies among modules* to the *number of all possible dependencies among modules*. Ideally, the value of RoC would be low, meaning that only a small part of all possible dependencies among modules is actually utilized—making it less likely that faults, failures, or errors introduced by changes or additions to modules will propagate across modules.

The second metric we selected is a variant of Cumulative Component Dependency (CCD) [38] which is the sum of all outgoing dependencies for a component. For Java-9 modules, outgoing dependencies are *requires* and *uses* dependencies of each module. The specific variant we used is Normalized CCD (NCD), which is the ratio of CCD for each module to the total number of modules. Ideally, the value of CCD, or NCD, is low, indicating lower coupling and better encapsulation.

Table 5.3 presents the amount of RoC and NCD change in 28 Java-9 applications with inconsistent dependencies. Across all 28 applications, the amount of RoC is reduced by an average of 25.34%, and up to 80.56%. The amount of NCD is also reduced in 15 applications by an average of 20.73%, and up to 79%. These results indicate that DARCY can successfully enhance the encapsulation of Java-9 applications by a significant amount.

5.5 RQ5: Software Bloat

To answer this research question, we measured the runtime memory needed by each application before and after DARCY's repair phase. Recall the fact that in Java 9, with the JDK being modularized, we are able to create a lightweight custom Java Runtime Environment (JRE), reducing software bloat. More specifically, the size of a custom JRE may be reduced after a repair if the application has inconsistent dependencies of type *requires JDK* (Equation 3.2 of Section 3).

Table 5.4 shows reduction of software bloat in terms of runtime memory size of affected applications after removing inconsistent *requires JDK* dependencies. According to the results, the reduction is about 14% in 6 applications, and up to 55%. Such results are particularly substantial for deployment and scalability goals in microservices or IoT devices that contain very little memory.

5.6 RQ6: Performance

As described in Section 4, Darcy builds on three tools, Classycle[26], Soot [68], and ANTLR [4]. As a result, to assess Darcy's performance we answer RQ5 in terms of these three underlying tools' execution time.

Table 5.5 describes the average execution times for Darcy. Results for Classycle are shown separately from results for other components since the execution time is dominated by Classycle. On average, Darcy takes under 9 seconds for any system to execute, which is highly time efficient for both detection and repair.

Table 5.3: Results for Encapsulation Improvement

Application Name	# Directives (before)	RoC % Change	NCD % Change
sense-nine	31	29.03	16.00
number-to-text	11	9.09	-
vstreamer	25	16.00	20.00
jigsaw-resources	5	20.00	-
JavaUtils	36	80.56	79.31
BunnyHop	28	60.71	25.00
java9-modules	5	20.00	-
jwtgen	13	15.38	8.33
project-constantin	9	55.56	20.00
java-spi-example	27	15.38	5.56
codersonbeer-app	13	30.77	12.50
rahmnathan-utils	14	50.00	12.50
auto-sort	13	7.69	-
java9-demo	10	10.00	-
java9-modules-tlb	12	8.33	-
java-9-lab	15	6.67	-
meetup-16	14	42.86	-
java-9-bookstore	17	17.65	16.67
springuni-java9	6	50.00	40.00
java-9-modularity	11	9.09	12.50
java-9-spring-mvn	18	44.44	16.67
music-ui-start	15	26.67	10.00
java9-labs	10	40.00	-
practical-security	20	10.00	7.69
java9-junit	13	7.69	-
the-message	16	6.25	-
jigsaw-tst	11	9.09	-
TRPZ	19	10.53	-
Total # of Affected Systems (RoC)			28
RoC Reduction Avg.			25.34%
Total # of Affected Systems (NCD)			15
NCD Reduction Avg.			20.73%

Table 5.4: Results for Software-Bloat Reduction

Application Name	JRE Size (MB) (before)	(after)	Runtime Memory Reduction (%)
sense-nine	19.11	18.99	0.63
JavaUtils	39.24	30.66	21.87
BunnyHop	46.23	20.93	54.72
java-spi-example	41.40	38.9	6.04
rahmnathan-utils	15.61	15.60	0.12
practical-security	15.72	15.60	0.76
Avg. Memory Reduction			14.02%

Table 5.5: Results for Execution Time

Component	Avg. Execution Time (ms)
Class Dependency Analyzer (Classycle)	7428
Java Reflection Analysis	328
ServiceLoader Usage Analysis	315
Java Inconsistency Analysis	250
Repair	453
Total	8774

Chapter 6

Threats to Validity

In terms of accuracy, the main threat to internal validity is the risk of false positives or negatives of the static analysis tools used in the implementation. False positives or negatives in the results of the static analysis tools may cause DARCY to miss some inconsistencies in the detection phase or report false inconsistencies, which may lead to compilation errors or harming functionality of the application after the repair phase. Since DARCY takes Classycle's results as an input for the Java inconsistency analysis, it inherits all of Classycle's limitations. The accuracy of detecting the inconsistent dependencies is affected by the accuracy of the static analysis tool we use. However, Classycle has been used and in development for over 11 years and leveraged by other state-of-the-art tools for software architecture and antipattern analysis [59, 72, 20, 41, 28, 42, 19]. A similar threat to internal validity exists for our use of Soot; however, Soot is a widely used[33, 39] and actively maintained framework [7] for static analysis of Java programs. We further manually determine whether every identified inconsistency is correct to ensure that any unforeseen issues with underlying static analyses do not compromise DARCY's accuracy.

One of the main threats to external validity is the selection and number of Java applications

in the evaluation dataset. To mitigate this threat, we selected open source Java-9 applications from many developers and about a hundred pages of search results on GitHub, one of the largest and most widely used open-source repositories online. Another threat to external validity is whether the types of inconsistencies we identify comprehensively cover those that may exist. To alleviate this threat, we considered the architectural inconsistencies based on all types of module directives defined in Java 9.

Darcy's evaluation on only one programming language, i.e., Java, is another threat to external validity. This threat is alleviated by the fact that Java is one of the most widely used languages in the world [9, 8]. Furthermore, the general idea behind Darcy can be extended to any other languages with modular programming constructs that utilize provides and requires interfaces advocated by software architecture-based development and design [70, 49, 40].

Chapter 7

Related Work

The most closely related literature to Darcy bridges the gap between software architecture and implementation. There are a variety of different types of strategies to address this issue: focusing only on the descriptive architecture by reverse engineering it; obtaining the descriptive architecture and the prescriptive architecture, followed by checking their conformance; ensuring that early in the software lifecycle that the descriptive and prescriptive architectures conform by providing architectural constructs in code; and approaches that ensure conformance of the descriptive and prescriptive architecture from the beginning and into maintenance.

Many approaches address the architecture-implementation mapping issue by ignoring the prescriptive architecture and simply trying to obtain the most accurate descriptive architectures possible [25, 35, 42, 34, 29, 28, 50, 58, 21, 53]. A large number of these approaches rely on software clustering to determine components from implementations [62, 47, 25, 35, 14].

A series of approaches detect inconsistencies between architecture and implementation by reverse engineering the descriptive architecture from the code and comparing it with the prescriptive architecture[51, 48, 69, 52, 66, 11, 71, 56, 24, 57, 37, 18, 36, 32]. Murphy et al.

introduced the software reflexion method which helps an engineer compare prescriptive and descriptive architectures in a manual manner [51]. A number of these approaches extend the reflexion method with automated architecture recovery techniques [37, 18, 36].

Other approaches provide implementation-level constructs that represent architectural elements (e.g., customizable programming-language classes representing components) that help ensure architectural conformance from a forward-engineering perspective [12, 17, 31, 43, 13, 54, 61, 67]. Many of these approaches support various notions of software architectural connectors or interfaces, rather than just components.

Certain approaches achieve architecture-implementation mapping from both a forward-engineering (e.g., code generation) and reverse-engineering perspective, i.e., round-trip engineering [74, 73, 63]. 1.x-way mapping [74] allows manual changes to be initiated in the architecture and a separated portion of the code, with architecture-prescribed code updated solely through code generation. 1.x-line mapping [73] extends 1.x-way mapping to product-line development. Song et al. [63] introduce a runtime approach for architecture-implementation mapping from a roundtrip-engineering perspective.

Darcy is the first approach that supports architectural-implementation conformance checking in a mainstream programming language using architectural constructs built directly into the programming language by its creators. Furthermore, our approach includes repair of non-conforming architectures, rather than just determining inconsistencies. Darcy is the only approach for architecture-implementation mapping that focuses on software bloat and attack-surface reduction.

The module system has been recently introduced in Java, and the only existing framework similar to JPMS is OSGI [2]. The major differences between OSGI and JPMS are as follows. OSGI was not able to modularize the JDK, preventing the construction of customized runtime images with a minimized JDK, which JPMS enables. Additionally, OSGI cannot

handle reflective access to modules' internal packages. Similar dependency-analysis facilities for OSGI are limited to removing unused dependencies of type import, which represents the require dependency, and cannot cover the other 7 types of inconsistencies in JPMS applications previously introduced in section 3. Therefore, there is no similar facility for OSGI that repairs all types of inconsistent dependencies as Darcy does.

Chapter 8

Conclusion

This paper formally defines 8 types of architectural inconsistencies in Java-9 applications and introduces DARCY, an approach for automatic detection and repair of these types of inconsistencies. DARCY leverages custom static analysis, state-of-the art static analysis tools, and a custom parser generator in its implementation to effectively detect and robustly repair architectural inconsistencies. The results of our evaluation indicates a pervasive existence of architectural inconsistencies among open source Java-9 applications. According to our experiment, DARCY's automatic repair results in a significant reduction of the attack surface, enhancement of encapsulation, and reduction of memory usage for Java-9 applications. In the future, we aim to expand DARCY to other programming languages and improve it to (1) provide architectural visualization and (2) be used as a plug-in for Java Integrated Development Environments (IDE) which helps developers avoid architectural inconsistencies when developing Java applications.

Bibliography

[1] Oracle Corporation. API specification for the Java Platform, Standard Edition: Class ServiceLoader. `https://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html`.

[2] OSGI Alliance. `https://www.osgi.org/developer/specifications/`.

[3] Project Jigsaw. `http://openjdk.java.net/projects/jigsaw/`, 2017.

[4] ANTLR. `http://www.antlr.org`, 2018.

[5] GitHub. `https://github.com`, 2018.

[6] ServiceLoader (Java SE 9 & JDK 9). `https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html`, 2018.

[7] Soot GitHub Issue. `https://github.com/Sable/soot/issues`, 2018.

[8] The State of the Octoverse 2017. `https://octoverse.github.com/`, 2018.

[9] TIOBE Index for August 2018. `https://www.tiobe.com/tiobe-index/`, 2018.

[10] Darcy web page. `https://seal.ics.uci.edu/projects/darcy/index.html`, 2019.

[11] M. Abi-Antoun, J. Aldrich, D. Garlan, B. Schmerl, N. Nahas, and T. Tseng. Improving system dependability by enforcing architectural intent. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[12] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM.

[13] J. Aldrich, C. Omar, A. Potanin, and D. Li. Language-based architectural control. In *Proceedings of the International Workshop on Aliasing, Capabilities and Ownership (IWACO)*, pages 1–11, 2014.

[14] R. A. Bittencourt and D. D. S. Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 251–254. IEEE, 2009.

[15] E. Bouwers, A. van Deursen, and J. Visser. Quantifying the encapsulation of implemented software architectures. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 211–220. IEEE, 2014.

[16] L. C. Briand, S. Morasca, and V. R. Basili. Measuring and assessing maintainability at the end of high level design. In *Software Maintenance, 1993. CSM-93, Proceedings., Conference on*, pages 88–87. IEEE, 1993.

[17] M. Caporuscio, H. Muccini, P. Pelliccione, and E. Di Nisio. Rapid system development via product line architecture implementation. In *International Workshop on Rapid Integration of Software Engineering Techniques*, pages 18–33. Springer, 2005.

[18] A. Christl, R. Koschke, and M.-A. Storey. Equipping the reflexion method with automated clustering. In *Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[19] E. Constantinou, G. Kakarontzas, and I. Stamelos. Open source software: How can design metrics facilitate architecture recovery? *arXiv preprint arXiv:1110.1992*, 2011.

[20] E. Constantinou, G. Kakarontzas, and I. Stamelos. Towards open source software system architecture recovery using design metrics. In *2011 15th Panhellenic Conference on Informatics*, pages 166–170, Sept 2011.

[21] E. Constantinou, G. Kakarontzas, and I. Stamelos. An automated approach for noise identification to assist software architecture recovery techniques. *Journal of Systems and Software*, 107:142–157, 2015.

[22] P. Deitel. Understanding Java 9 Modules. `https://www.oracle.com/corporate/features/understanding-java-9-modules.html`, 2017.

[23] P. J. Deitel and H. M. Deitel. *Java 9 for Programmers*. Prentice Hall, 2017.

[24] J. A. Diaz-Pace, J. P. Carlino, M. Blech, A. Soria, and M. R. Campo. Assisting the synchronization of ucm-based architectural documentation with implementation. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) & 3rd European Conference on Software Architecture (ECSA)*, pages 151–160. IEEE, 2009.

[25] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[26] F.-J. Elmer. Classycle: Analysing Tools for Java Class and Package Dependencies. *How Classycle works*, 2012.

[27] J. Garcia, M. Hammad, and S. Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):11, 2018.

[28] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 486–496. IEEE Press, 2013.

[29] M. W. Godfrey and E. H. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *Proceedings of Second Symposium on Constructing Software Engineering Tools (CoSET'00)*. Citeseer, 2000.

[30] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith. The Java® Language Specification Java SE 9 Edition. Technical report, Technical report, Oracle Corporation. https:https://docs.oracle.com/javase/specs/jls/se9/html/index.html, 2017.

[31] J. Grundy. Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 10(06):713–734, 2000.

[32] A. Gurgel, I. Macia, A. Garcia, A. von Staa, M. Mezini, M. Eichberg, and R. Mitschke. Blending and reusing rules for architectural degradation prevention. In *Proceedings of the 13th international conference on Modularity*, pages 61–72. ACM, 2014.

[33] L. Hendren. Uses of the Soot Framework. `http://www.sable.mcgill.ca/~hendren/sootusers/`, 2018.

[34] I. Ivkovic and M. Godfrey. Enhancing domain-specific software architecture recovery. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 266–273. IEEE, 2003.

[35] R. Koschke. Architecture reconstruction. In *Software Engineering*, pages 140–173. Springer, 2006.

[36] R. Koschke, P. Frenzel, A. P. Breu, and K. Angstmann. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17(4):331–366, 2009.

[37] R. Koschke and D. Simon. Hierarchical reflexion models. In *null*, page 36. IEEE, 2003.

[38] J. Lakos. Large-scale c++ software design. *Reading, MA*, 173:217–271, 1996.

[39] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[40] K. Lau and Z. Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, Oct 2007.

[41] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 235–245. IEEE, 2015.

[42] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 69–78. IEEE Press, 2015.

[43] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, 2005.

[44] P. Manadhata and J. M. Wing. Measuring a system's attack surface. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2004.

[45] P. K. Manadhata, K. M. Tan, R. A. Maxion, and J. M. Wing. An approach to measuring a system's attack surface. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2007.

[46] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, (3):371–386, 2010.

[47] O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11), 2007.

[48] N. Medvidovic, A. Egyed, and P. Gruenbacher. Stemming architectural erosion by coupling architectural discovery and recovery. In *STRAW*, volume 3, pages 61–68, 2003.

[49] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Jan 2000.

[50] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic-centric approach for automating traceability of quality concerns. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 639–649. IEEE, 2012.

[51] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[52] L. O'Brien, D. Smith, and G. Lewis. Supporting migration to services using software architecture reconstruction. In *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, pages 81–91. IEEE, 2005.

[53] D. Qiu, Q. Zhang, and S. Fang. Reconstructing software high-level architecture by clustering weighted directed class graph. *International Journal of Software Engineering and Knowledge Engineering*, 25(04):701–726, 2015.

[54] A. Radjenovic and R. F. Paige. The role of dependency links in ensuring architectural view consistency. In *Seventh Working IEEE/IFIP Conference on Software Architecture*, pages 199–208. IEEE, 2008.

[55] M. Reinhold. JSR 376: Java Platform Module System. Technical report, Technical report, Oracle Corporation. http://cr.openjdk.java.net/ mr/jigsaw/spec/, 2014.

[56] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *ACM Sigplan Notices*, volume 40, pages 167–176. ACM, 2005.

[57] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th international conference on Software engineering*, pages 387–396. IEEE Computer Society, 1996.

[58] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, and N. Medvidovic. Recovering architectural design decisions. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018.

[59] M. R. Shaheen and L. du Bousquet. Quantitative analysis of testability antipatterns on open source java applications. *QAOOSE 2008-Proceedings*, page 21, 2008.

[60] K. Sharan. The module system. In *Java 9 Revealed*, pages 7–30. Springer, 2017.

[61] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE transactions on software engineering*, 21(4):314–335, 1995.

[62] M. Shtern and V. Tzerpos. Clustering Methodologies for Software Engineering. *Adv. Soft. Eng.*, 2012:1:1–1:1, Jan. 2012.

[63] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software*, 84(5):711–723, 2011.

[64] R. Taylor, N. Medvidovic, and D. E.M. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2009.

[65] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A component- and message-based architectural style for gui software. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 295–304, New York, NY, USA, 1995. ACM.

[66] J. B. Tran, M. W. Godfrey, E. H. Lee, and R. C. Holt. Architectural repair of open source software. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 48–59. IEEE, 2000.

[67] N. Ubayashi, J. Nomura, and T. Tamai. Archface: a contract place where architectural design and code meet together. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 75–84. ACM, 2010.

[68] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[69] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 122–132. IEEE, 2004.

[70] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, March 2000.

[71] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 470–479. IEEE, 2004.

[72] R. Yokomori, N. Yoshida, M. Noro, and K. Inoue. Extensions of component rank model by taking into account for clone relations. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 3, pages 30–36. IEEE, 2016.

[73] Y. Zheng, C. Cu, and R. N. Taylor. Maintaining architecture-implementation conformance to support architecture centrality: From single system to product line development. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(2):8, 2018.

[74] Y. Zheng and R. N. Taylor. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In *Proceedings of the 34th International Conference on Software Engineering*, pages 628–638. IEEE Press, 2012.

Appendix A

Artifacts

The artifacts are available for download from the following link:

`https://github.com/negarq/darcy-artifact`

A.1 Publication

This thesis has been published in 41st International Conference on Software Engineering (ICSE 2019)

- Negar Ghorbani, Joshua Garcia, Sam Malek, "**Detection and Repair of Architectural Inconsistencies in Java**", International Conference on Software Engineering (ICSE), Montreal, QC, Canada, May 2019. (21% acceptance rate)