

AN ADAPTER-AWARE, NON-INTRUSIVE DEPENDENCY INJECTION FRAMEWORK FOR JAVA

MASTER'S THESIS

Author:


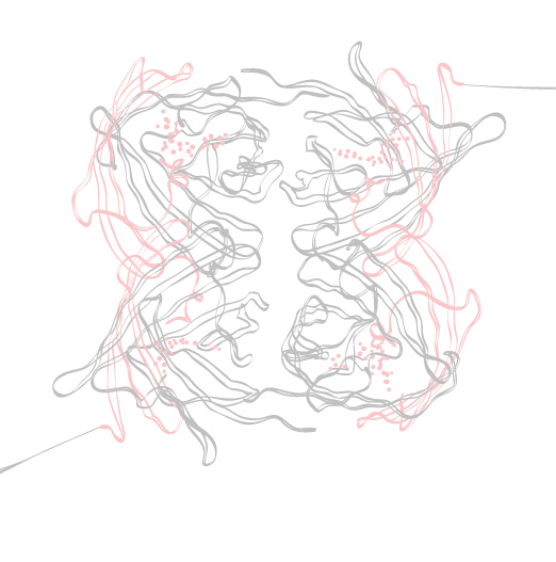
BSc. Arnout ROEMERS

Committee:

Dr. Ing. Christoph BOCKISCH

MSc. Kardelen HATUN

Dr. Ir. Lodewijk BERGMANS



May 16, 2013

Abstract

Contemporary software systems are constructed by composing different software components. In strongly typed Object-Oriented Programming languages, it is common to encounter type incompatibilities between separately developed software components one desires to compose. Using the Adapter pattern to overcome these type incompatibilities is only an option if changing the source code of the software components is feasible, as references from objects to other objects are oftentimes hard-coded. The concept of Dependency Injection (DI) is aimed at mitigating the issue of hard-coded references. However, current implementations of DI are intrusive in ways that component developers need to foresee future use cases. As it is impossible to foresee all future use cases, composing type incompatible components remains difficult.

This thesis presents our efforts towards a solution for enabling compositions entailing unforeseen use, without changing any source code. This is achieved by combining the Adapter pattern and DI in a certain way. Our solution increases reuse and loose coupling in a non-intrusive way for newly developed components, but also for (legacy) components that were not developed with certain reuse in mind. Our research comprises a validation of our solution concerning above composition problems, and the discovery of limitations and future improvements. This was done by defining and evaluating goals and requirements and by implementing a proof-of-concept of our conceptual solution. Using this proof-of-concept implementation, the validation includes a comparison with other (semi-)solutions and an evaluation of how it would be applied to real-world projects. The validation shows that in many situations our solution indeed solves above composition problems.

Acknowledgements

There are several people I would like to thank, for their support during the research and during the writing of this thesis.

First of all I would like to thank Kardelen Hatun, my direct supervisor, whos ability to quickly understand my brain dumps and to add great ideas to them, inspired me a lot. She showed me whether I was on the right track and she kept me going. The sessions with my second supervisor, Christoph Bockisch, were good times as well. Our shared preference to be precise in wording has hopefully contributed to the quality of this thesis and the work it describes. I think the three of us have found a nice way of working together and I can honestly say that it was a pleasant experience.

I would also like to thank Robert Dyer from Iowa State University, for his cooperation in using their tooling, in order to find real-world projects for the validation part of our research.

Next, I would like to thank my partner, Joline van Beek, for her unconditional support. If it were not for her persistence in that I should persue my wish to finish my master, this research would not have been performed altogether. I would also like to thank my parents, who have also played an important role in getting me where I stand now, by believing in me, and of course for giving me the Commodore 64 that started it all. Lastly, I would like to thank my brother, Stephan Roemers, for his interest in my study and his ability to shed a different light upon a topic, and my uncle Roelof Roemers, as he took the time to discover the world of programming and science with me during my youth.

Contents

1	Introduction and motivation	9
1.1	Motivation of the research	10
1.1.1	Background information	10
1.1.2	Problem: Incompatible types	11
1.1.3	Problem: Hard-coded dependencies	12
1.1.4	Problem statement	14
1.2	Conceptual solution	15
1.2.1	Our conceptual solution	15
1.2.2	Goals for our implementation	16
2	Gluer: A Proof-of-concept implementation	19
2.1	Semantics of Gluer	20
2.1.1	Adapters	21
2.1.2	Associations	23
2.1.3	Adapter resolution	25
2.2	Running the tool	33
2.2.1	Input files	33
2.2.2	Modes	33
2.3	Checks	35
2.4	Implementation and extensions	37
2.4.1	Implementation language: Clojure	37
2.4.2	Parsing	38
2.4.3	Multimethods	38
2.4.4	The Java agent	39
2.4.5	Relation with Aspect-Oriented Programming	40

3	Validation and findings	41
3.1	Validating Gluer's requirements	41
3.1.1	Comparison with compositions in real-world projects	42
3.1.2	Theoretical use case implementation	46
3.1.3	Overall requirements validation	49
3.2	Adapter selection semantics validation	51
4	Discussion	53
4.1	Open issues and limitations	54
4.1.1	Expressiveness of DSL	54
4.1.2	Concurrent use with other injection frameworks	55
4.1.3	Final classes as the type of the injection point	55
4.1.4	When to inject into a field	55
4.2	Future improvements	56
4.2.1	External improvements	56
4.2.2	Internal improvements	57
4.3	Related work	57
4.4	Conclusion	58
A	Boa results to HTML documents script	63
B	Validation source code	67
B.1	Hard-coded implementation	67
B.2	Google Guice implementation	74
B.3	Gluer implementation	77
B.4	Adapter resolution validation	79
C	Implementation details	81
C.1	Parsing	81
C.1.1	Simple example	82
C.1.2	Non-terminals as terminals	82
C.1.3	Parse errors	83
C.1.4	Whitespace	84

C.2 Adapter registry	84
C.3 Precedence relations	85
D List of acronyms	87

Chapter 1

Introduction and motivation

This chapter describes some common issues in composing Object-oriented components, including well-known concepts that solve them. Subsequently, the project's research goal to improve upon this, our solution towards this goal and the initial requirements for this solution are described.

§

One of the advantages that Object-Oriented Programming (OOP) has over other programming paradigms like the imperative paradigm, is the ease of reusing previously developed objects and components [Meyer, 1987]. Reuse is gained by having loosely coupled classes and components, together with privately encapsulated implementation logic. Reuse is also increased if a class or component is extensible, or “configurable”, for use cases slightly different than its initial use case. Still, it is not possible to foresee every future use case and extensions, and every design has to make trade-offs in this respect as well. Therefore, composing or binding separately developed components is not always trivial, even when OOP is used.

Nonetheless, many approaches exist for dealing with composition problems and for increasing the opportunities of reuse. Some of them are discussed in this thesis, and it is an active field of research. These approaches can be in the form of simple design patterns, but also encompass entire frameworks. We will see examples of both in this thesis. This thesis discusses the problem with the current approaches and it presents our solution for mitigating the problem and thereby improving software compositions.

The scope of this thesis is to specifically discuss composition issues and concepts for OOP languages that have closed classes and are statically typed, such as Java. Object-oriented systems that have open classes and/or are dynamically typed, such as the Common Lisp Object System (CLOS) [Kiczales et al., 1991], are not part of this research. How the issues and solutions that are described relate to those other object

systems, is therefore out of scope. Furthermore, this thesis focuses on composition concepts on the level of OOP itself, e.g. types, fields and methods. Higher level compositions through for example network connections (e.g. REST, pipes or queues), intentional use of runtime containers (e.g. OSGI) or alike, are not discussed. We have chosen this scope in order to focus on the foundational level of programming, i.e. its constructs. The design and possibilities of these foundational constructs trickle up to the higher-level compositions realised by them.

The structure of this thesis is as follows. The remainder of this chapter will give some background information by introducing common composition issues and techniques for overcoming these issues. After these concepts have been introduced, we show how most of the solutions are limited, and we define the problem statement. Next we present our conceptual solution to the problem statement, how it solves the problems, and the goals and requirements for a concrete implementation of it.

Chapter 2 contains a detailed description about our concrete proof-of-concept implementation. It discusses how it fulfills the goals and requirements we defined. It also describes how it works, i.e. the semantics and usage, and some important design decisions. Chapter 3 contains the validation and evaluation of our conceptual solution and it's goals using the proof-of-concept and discusses the limitations of the current work. Chapter 4 wraps things up by discussing possible future extensions and related work.

1.1 Motivation of the research

In order to motivate our research and define our problem statement, we first give some background information on composition problems. The next subsection describes two example composition issues that may arise in OOP, including an approach on how to overcome them. Subsequently we can describe the problems with the current approaches and we present our problem statement.

1.1.1 Background information

To make clear on what composition issues we want to improve, we will explain them by outlining a typical example situation. This example will be used throughout the entire thesis.

Consider developing a software application. This application uses a component that offers an Application Programming Interface (API) to perform logging. Such an API consists of classes and objects, with methods one can call. The application also uses *another* component. The function of this other component is not relevant to our discussion per se, but let us say it is a component that retrieves data over the HyperText Transfer Protocol (HTTP). This HTTP component is also using a logging

component itself, but different from the one the application uses. The situation is schematically visualised on the left side in Figure 1.1.

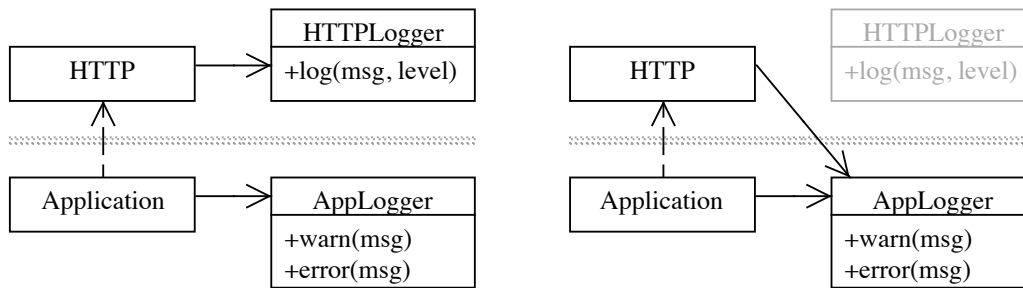


Figure 1.1: Left: the situation — Right: the preferred structure.

Typically one does not want to have multiple active logging components in a single application. Mitigating this issue involves having the HTTP component use the application logger (AppLogger in the figure) instead, as shown on the right hand side of Figure 1.1. The other way around, i.e. having the application use the logger used by the HTTP component, makes less sense, especially if you consider more components in the application using yet another logger. Also note that this problem still exists when one uses an abstraction layer for loggers, such as the SL4J library. In such a case one can think of it that one abstraction layer is not compatible with the other, so one wants to use a single one.

At least two problems might occur when working towards the preferred structure, but concepts exist to solve them. Both problems, that of incompatible types and that of hard-coded dependencies, are explained in the next subsections, along with approaches that solve them.

1.1.2 Problem: Incompatible types

One of the problems that arise when having the HTTP component use another logging component, is that of incompatible types. As noted above, an API consists of classes and objects with methods. More likely than not, the classes of the logging APIs are type incompatible with each other. Probably the method signatures and their semantics differ as well. We could update the API of the application logger, so it becomes type compatible with the HTTP logger. Because we would need to do this for every other logger and it would make our logger overly complicated, it does not make much sense to do so, even *if* the source code is available.

Solution: The Adapter pattern

If one wants to use a class in a place where another type is expected, a common solution is to use the Adapter pattern [Gamma et al., 1995]. As the book puts it, the “*Adapter*

lets classes work together that couldn't otherwise because of incompatible types.”. The Adapter pattern places an intermediary class, the adapter, between the “caller” and the type incompatible “callee”. This adapter class aggregates the callee, now called the *adaptee*, and is type compatible with what the caller expects. It “translates” the method calls on it to calls the callee understands. Figure 1.2 shows a class diagram using the Adapter pattern.¹

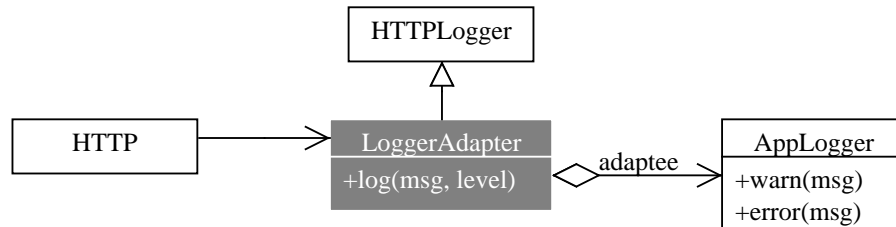


Figure 1.2: Structure of the Adapter pattern.

Regarding our situation with the two loggers, one could adapt the application logger to something that is type compatible with the HTTP logger, like the `LoggerAdapter` class in the figure, which solves the type incompatibility issue.

1.1.3 Problem: Hard-coded dependencies

However, another more complex problem remains. We want to swap the HTTP logger with the adapted application logger, but chances are that the instantiation and use of the HTTP logger is hard-coded. In other words, the dependency is created in the HTTP component, without any outside influence on exactly *what* is created. In this case, it is difficult to perform the swap without changing the source. Of course, the HTTP component might already have been developed with swapping possibilities in mind, i.e. influencing dependencies, by using a suitable abstraction. The abstraction we will discuss in this regard is the concept of Dependency Injection (DI).

Solution: Dependency injection

Dependency Injection, also known as Inversion of Control (IoC), is a design pattern that takes the control of which concrete dependency to use outside of the dependent component [Fowler, 2004]. In our example, this means that which concrete `HTTPLogger` to instantiate and use in the HTTP component can be controlled externally. We will first explain in more detail what DI is, and subsequently review how this can be applied in our scenario.

¹We use the *object* adapter pattern, not the *class* adapter pattern that is also discussed in the “Gang of Four” book [Gamma et al., 1995].

First let us look at what problem DI actually solves. In listing 1.1 one sees that the dependent, class HTTP in this case, is in charge of what logger implementation (the dependency) it uses.

Listing 1.1: Hard-coded dependency.

```
1 public class HTTP {
2     private HTTPLogger logger = new HTTPLogger();
3     ...
4 }
```

The logger is created within the class itself. To change the logger type, which is what we want in our example, the class source would need to be changed. A construction like this can be problematic in all kinds of scenarios. For example, changing a dependency to a so-called *mock* version for testing purposes cannot be easily automated.

DI solves this problem. By defining an interface for the dependency and having concrete implementations of that interface, the instantiation of such a concrete implementation can then be done outside of the depending component. For illustrational purposes, we first show an easy way of doing this, where it must be noted that this is not the recommended way in larger software systems. That said, an easy way to supply the depending component with a concrete dependency, is by passing it through its constructor. This yields code like in listing 1.2.

Listing 1.2: Constructor Dependency Injection.

```
1 // Structure
2 public HTTP {
3     private Logger logger;
4
5     public HTTP(Logger logger) {
6         this.logger = logger;
7     }
8     ...
9 }
10
11 // Initialisation and use.
12 Logger log = new HTTPLogger();
13 HTTP http = new HTTP(log);
```

The last two lines in the listing now control which concrete `Logger` is used in the `HTTP` class. The example is a very simple implementation, but it shows the essence of DI. However, as noted before, this way of applying DI is not recommended for projects of even a moderate size, let alone large projects. If the depth of classes depending on other classes is high, implementing DI using only the standard language features leads to unmanageable code of passing through the dependencies. Using the *Factory*

Method design pattern [Gamma et al., 1995] in this regard mitigates this issue, as the dependants then have a central point to retrieve their dependencies from, but this can still lead to tightly coupled code as well. Therefore, DI is often implemented and used by way of a framework. Typical frameworks in the Java environment that support DI are JNDI, Spring and Google Guice.

Going back to our situation, if the HTTP component uses some kind of DI, then injecting the adapter that wraps the application logger would be easy and sufficient for our goal. But more often than not, the dependency is hard-coded, thus making injecting the adapter a lot more difficult, and definitely more complicated than the two simple patterns we have discussed.²

1.1.4 Problem statement

The problem we see is, also for our example case, that most of these approaches need to be applied from the beginning of the development of an application or component. This way they become an intertwined part of the software. This is also the case with the Adapter pattern and DI. While this intertwined nature is an undesirable property on its own, two other major problems arise due to it.

The first problem is that the composition approach of one component may not be compatible with the composition approach of another component, which makes composing the two difficult. Due to the intertwined nature of these approaches one cannot simply switch to another. The second problem is that by having the composition approach as an integral part of the software, one still has the issue that one cannot foresee every future use case and “composition point”.

There are composition approaches that can be “tagged on” later in the development. For example with Aspect-Oriented Programming (AOP). The big advantage of these approaches over those that are intertwined, is that one separates the composition logic from the core logic. They are *non-intrusive*. This means one can design and implement a component, without needing to guess every use case and extension point up front, such that one can focus on the core logic. However, the disadvantage is that these approaches are often complex, whereas the concepts like the Adapter pattern and DI are relatively simple.

One can say that composition approaches are part of the foundation on which software is build. Simplicity is important for any foundation. The simpler the foundation, the better a developer can easily wrap his head around it, and the less the foundation gets in the way of writing complex logic. Therefore, we think that composing should be simple.

In conclusion, the problem is that we need to find a composition technique that is

²There exist Aspect-Oriented Programming (AOP) techniques that are able to do such an injection. More on AOP and its relation with our research can be found in Section 2.4.5

both non-intrusive, i.e. not intertwined in the software component, *and* simple, like the Adapter pattern. A solution for this should have the best of both worlds, thereby improving reuse and loose coupling for, possibly separately developed, Object-Oriented software components.

1.2 Conceptual solution

We have defined some requirements for a solution to our problem. These requirements are extracted from the problem statement, and also narrow our research scope even more. A solution to our problem must ...

[req 1] ... have capabilities for objects to have an existing field refer to other, externally declarable, possibly type incompatible objects, without needing to change any source code.

[req 2] ... keep it simple. The solution should be simple, intuitive and easy to fathom from a user's perspective, i.e. it must be possible to quickly have a complete mental model in one's head of what is happening under the hood. Also, the steps to perform for incorporating the solution should be kept to a minimum and simple.

[req 3] ... be worthwhile to use the solution. The work it saves can be quantified in for example lines of code that need to be written or the amount of testing required to reach confidence about the robustness of the composition.

These requirements will be revisited when we validate our conceptual solution from the next subsection.

1.2.1 Our conceptual solution

With the problem statement and the requirements in mind, we have defined a specification that encompasses a "Non-intrusive, Adapter-aware Dependency Injection framework". This section describes what it does and how we think it is a solution to the problem statement.

Both the concept of Adapters and DI have already been explained in the former section. What is unique about our solution is how it combines both concepts, while conforming to the requirements we have set. The solution needs two input values. The first is a registry of adapter classes and the second is one or more statements that declare dependency injections. Both need to be defined in an external fashion, i.e. outside of the components. This avoids intertwining and the need for any change in the components' source.

Each injection statement declares *what* is injected and *where* it needs to be injected. It is through these injection statements that one composes components in an application. The framework takes these injection statements and transforms the application in such a way that these injections are performed when the application is run, effectively glueing the components together. However, one of the strengths of our solution lies in what happens when the type of what is injected is not compatible with where it is injected. In such a case, the framework automatically chooses and uses an adapter from a registry of adapters. Choosing an adapter follows defined resolution semantics, i.e., the semantics describe how the most suitable adapter is chosen from the registry of available adapters.

While the solution we have described so far may already solve our problem, especially for the requirements [req 1] and [req 2], there is more. The injection statements we envision lend themselves well to be written in a *declarative* Domain-Specific Language (DSL). Due to the declarative nature of such a DSL, the framework is able to check a lot of things before the application is run. For example, it can check whether there is a suitable adapter available for each injection statement. Of course, for this to work, the framework needs to be able to apply static analysis on the application, and have access to the adapter registry. Still, having such an analysis contributes to the robustness of our solution with respect to requirement [req 3].

Finally, since the injection statements in the DSL we speak of do not necessarily inject the specified dependency directly into the injection point, we call these statements *associations* throughout the thesis. We say that we *associate* an injection point with a dependency.

1.2.2 Goals for our implementation

We now have a conceptual solution to our problem. When realising the framework we have described, i.e. developing a concrete implementation, we think it is important to keep some supplementary goals in mind. For our implementation we have defined the goals below. These goals are in addition to the requirements, and can be seen as quality requirements.

[goal 1] An implementation should be able to check for errors that may arise from using the solution. The more pre-runtime checks within this scope, the more developer can be confident that no issues are encountered when the application is run. Note that this is not about errors that lie beyond that of what the solution adds to the composite application.

[goal 2] An implementation should automate the choice of which adapter to use as much as possible, with clear semantics. Such automation could be accompanied by ways of influencing this automation.

[goal 3] For association statements, many choices can be made. For example, what kind of injection points it supports and how one selects these points. Therefore, the framework should be simple to be extended with new selection constructs in association statements.

The next chapter will describe our implementation of our conceptual solution, which has been designed with above goals in mind. Of course, these goals may be applicable to other implementations of our solution as well.

Chapter 2

Gluer: A Proof-of-concept implementation

*This chapter presents a proof-of-concept implementation of our approach.
Our aim is to show how we think the theoretical solution from the first
chapter can be implemented.*

§

This chapter describes our proof-of-concept implementation of our solution from the first chapter, which we have called *Gluer*. This framework is made for the Java language and we have developed it to enable us to reason about and validate our solution, in addition to show how such a framework can be realised.

In summary, the Gluer framework takes Adapter classes and association statements as input and this is used for composing components. Both the input and how this composition works will be discussed in the next sections, but it is good to know up front that the Gluer framework has two modes of execution. These are the *checking* mode and the *runtime* mode. The checking mode uses *static analysis* to check the input given to the framework, in order to signal possible issues that may arise during runtime mode. It is the runtime mode that actually performs that what is given as input. A more detailed explanation of the execution modes is given in Section 2.2.2.

The rest of this chapter is structured as follows. *What* exactly is performed is defined by the semantics of Gluer. Since our implementation does not involve a lot of (new) syntax, the semantics are leading in the first part of this chapter (section 2.1) and the input for the Gluer framework is covered as we go. The semantics are the same in both modes of execution.

Another key point of our conceptual solution is this reporting of possible composition issues using the static analysis in the checking mode. Therefore, Section 2.2 has a more detailed explanation on the two modes of execution in Gluer and Section 2.3 covers the static checks we have implemented. Subsequently, Section 2.4 describes some of the implementation details regarding the extensibility of the framework. More in depth details on the implementation can be found in appendix C.

2.1 Semantics of Gluer

To first get an idea of where the Gluer framework is positioned with respect to the inputs and the application it is applied to, Figures 2.1 and 2.2 show schematic overviews. The difference between the two figures has to do with the mode of execution Gluer is in. Note that the red, underlined components are what the Gluer framework contributes.

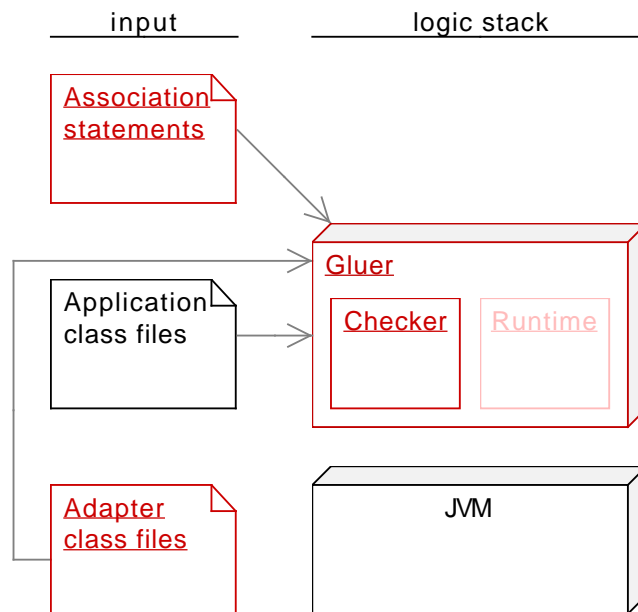


Figure 2.1: The position of Gluer and the flow of information, at checking mode.

Figure 2.1 shows the flow of information, as denoted by the arrows, when Gluer is run in checking mode. In this mode, Gluer acts as an application itself on top of the Java Virtual Machine (JVM). One sees that the association statements are read directly by Gluer, but more importantly, the class files are read directly by Gluer as well. This way, the aforementioned static analysis can be performed without actually loading the classes in the JVM.

Figure 2.2 shows how the flow of information changes, and also how the position of Gluer changes, when the tool is in the runtime mode of execution. In this mode, one can think of Gluer as sitting *in between* the application and the JVM.

The application class files are now loaded by the JVM as demanded by the application, but one sees that these classes go through the Gluer framework while they are loaded. It is here where the classes are prepared for injections. These injections are performed with help of the Gluer runtime component, which is why a link between the application and the Gluer runtime exists. The injections may need Adapters, which is why those are also loaded into the JVM and end up in the application. Note that the class files are still read directly by the Gluer framework as well, for the static analysis.

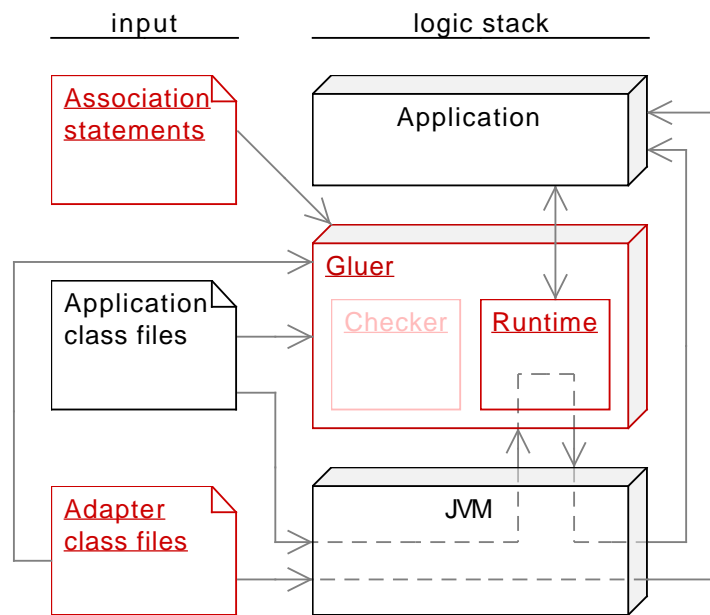


Figure 2.2: The position of Gluer and the flow of information, at runtime mode.

One final note on these overviews, is that the checker and runtime components have been depicted as parts of the Gluer component. This is because Gluer is a single component tool, although capable of running in different modes of execution, and the checker and runtime share a lot of implementation logic.

2.1.1 Adapters

As discussed in the first chapter, the key point of our solution is to have automatic adaptation for type-incompatible injections. For this, the Gluer framework has a registry of adapter classes available, as is also depicted on the left side of Figure 2.1/2.2. We have chosen for adapters to be plain Java classes. This choice fits our goal of minimising the incidental complexity of our framework, since there is no new syntax to be learned and no expressiveness is lost when developing the adapters. This way, the developer is flexible in defining the adapters as he or she sees fit. It is the Gluer framework that instantiates these classes automatically when needed.

Although the adapter classes are plain Java classes, they do need to have some properties in order to be discoverable and usable by Gluer. These properties are:

1. They need to be tagged with the `@Adapter` annotation. This way, the framework recognises it as an adapter and will register it into the adapter registry.
2. The types (classes and interfaces) it extends or implements determine where the adapter can be used, i.e. where it can be injected. There are no restrictions on what the adapter extends or implements. This means an adapter extending or implementing nothing is possible as well, but it would only be applicable for injection points where a `java.lang.Object` is expected.
3. The constructors with a single non-primitive argument determine what an adapter can “adapt” from. The single argument to such a constructor is the *adaptee*, as explained in Section 1.1.2. At least one such constructor is required. More are allowed and are well supported, but it is advised that the adaptees are related. Otherwise it may be better to implement multiple adapters.
4. Furthermore, there are some restrictions on the modifiers of an adapter class. The class needs to be declared public. If the class is a member of another class, it needs to be declared static as well. The class cannot be abstract. These properties are required, so that the class can be instantiated by the framework.

While research is ongoing on adaptation of different cardinalities, e.g. adapting multiple objects to a place where a single object is expected, with Gluer we focus on one-to-one relations. The second property already hints at this; the adapters for Gluer are only suitable for these one-to-one relations. Note that a collection data structure, such as a `List`, is a single object as well.

Taking our scenario from Chapter 1 as an example, we need to define an adapter that extends the `HTTPLogger` class and takes an `AppLogger` as its adaptee. This results in the following Java code:

Listing 2.1: Example adapter class.

```
1 import gluer.Adapter;
2
3 @Adapter
4 public class App2HTTPLogger extends HTTPLogger {
5
6     private AppLogger adaptee;
7
8     public App2HTTPLogger(AppLogger adaptee) {
9         this.adaptee = adaptee;
10    }
11
```

```

12     public void log(String msg, int lvl) {
13         switch (lvl) {
14             case 0: adaptee.warn(msg); break;
15             case 1: adaptee.error(msg); break;
16         }
17     }
18 }

```

Considering the properties for adapters, above listing defines a class that is recognised as an adapter due to the `@Adapter` annotation on line 3, that is type compatible with an `HTTPLogger` due to the extension on line 4 and takes an `AppLogger` as its adaptee due to the constructor on line 8. The adapter class is registered into the adapter registry together with these properties.

2.1.2 Associations

Composing objects using Gluer is done with what we call *associations*. An association is a declarative statement that specifies *what* object is associated *where*. These associations are like injections, as they inject the *what* into the *where*. The difference is that there is a layer of indirection, i.e. the adapters. Injections imply that what is specified as the injectee is directly injected in the specified injection point. In our framework this is not necessarily the case. As we will see when discussing the adapter resolution (section 2.1.3), in many cases the injectee is adapted and it is the *adapter* that is injected. This indirection is where the strength of our solution lies.

An association statement is defined in a separate file, i.e. not in a Java source file, as shown on the top left side of Figures 2.1 and 2.2. Multiple of these statements can be defined in such a file and they have the following form:

```
associate <where> with <what>
```

Both the `<where>` and `<what>` can be specified using several kinds of selection statements. As associations are declarative, naturally these selection statements are declarative as well. Since our associations have a strong similarity with “normal” DI, we can see the `<where>` selection statement as the *injection point*. That what is injected we call the *associatee*.

In our proof-of-concept implementation, one `<where>` selection has been implemented:

field The `field <where>` selection takes one argument, namely a reference to an instance field of a class. The reference is a Fully Qualified Name (FQN) [Gosling et al., 2012, 6.5.5]. Using this clause means that whenever the specified class is instantiated, the specified field will have the, possibly adapted, associatee

injected into it. The field is allowed to have any modifier, except `static`, for it should refer to an *instance* field, or `final`. For example, one could have a field selection like:

```
associate field HTTP.logger with ...
```

The example means that whenever a new `HTTP` object is instantiated, the `logger` field of that instance will have the, possibly adapted, associatee injected into it.

Note that the choice of only supporting instance fields, and not static fields or other injection points, is no limitation of our concept per se. Gluer is intended as a proof-of-concept, and choosing a simple and common injection point fits our goal of having a working implementation we can reason about, without complicating things. Future additions in this respect are discussed in Chapter 4.

The `<what>` selection statements declare what object to associate with the injection point. Currently, there are several options for such a `<what>` selection statement implemented:

new The `new` statement takes a fully qualified class name as its argument. When using this selection, a new instance of the specified class is created each time the `<where>` statement triggers an injection. The class must have a public non-argument constructor, so it can be instantiated by the framework. For example:

```
associate field HTTP.logger with new AppLogger
```

This example instantiates a new `AppLogger` each time an `HTTP` object is created, and associates the `logger` field of this particular `HTTP` object with this particular `AppLogger`.

single The `single` `<what>` selection statement also takes a fully qualified class name as its argument. The difference with the `new` statement, is that a single instance of the specified class is reused each time the `<where>` statement triggers an injection. In other words, the Gluer runtime instantiates the class only once (the first time it is requested) and reuses it for every injection done by this association. The class must have a non-argument constructor, so it can be instantiated by the framework. An example association using the `single` statement is:

```
associate field HTTP.logger with single AppLogger
```

This example would instantiate an `AppLogger` object the first time an `HTTP` object is created, and associates the `logger` field of this particular `HTTP` object with this particular `AppLogger`. The next time an `HTTP` object is created, it associates the `logger` field with the *same* `AppLogger`.

retval The last currently supported <what> selection statement is the `retval` statement, which is short for “return value”. It means a call to a public, static, non-void method each time the <where> statement triggers an injection. The statement therefore takes a FQN reference to such a method as an argument, with the parameters for the method call at the end. These parameters can be any arbitrary Java expression. The return value of the method call determines what is associated with the injection point. The `retval` statement gives more expressive power, in case the former two <what> statements are not sufficient. An example:

```
associate field HTTP.logger with retval
    LoggerFactory.get(Config.isProduction())
```

Above example assumes the availability of a `LoggerFactory` class with a static method `get`. It is this method that is called each time an `HTTP` object is instantiated, and the `logger` field of it is associated with the return value of `get`. One sees that the result of a call to `Config.isProduction` is used as an argument, which may influence the return value of `get`. Note that the parameter expressions are evaluated again for each individual injection. This parameterisation is an example of the extra expressiveness the `retval` statement offers.

2.1.3 Adapter resolution

Now that we have seen the two basic building blocks for Gluer, the adapters and associations, we can discuss how they work together. An important characteristic of the selection statements in the associations is that the framework can determine the type, i.e. the class or interface, of them. More precise, it is only the *static* type that can be determined in the *checking* mode of execution. For the `new` and `single` selections the type is simply the referenced class. For the `retval` statement it can only determine the static type, which is the declared return type of the referenced method, though the actual return value at runtime may be a subtype. Of course, when Gluer is in *runtime* mode, the mode where the associatees are actually “live” objects, Gluer knows the actual type.

Having both types of the <where> and <what> parts of an association, and a registry of adapters, an algorithm for finding an eligible adapter is needed. We call this the *adapter resolution* and this section will describe it in detail. The adapter resolution is applied for associations that have incompatible types regarding the injection point (the <where>) and the associatee (the <what>). In the checking mode it is used for determining whether a suitable adapter is available, whereas in the runtime mode it is used for finding a suitable adapter for use in the injection. The algorithm is aimed at finding the *most suitable* adapter. This section describes the adapter resolution

process and what it means for an adapter to be most suitable.

The first step is to check whether an adapter is actually required. If the <what> type *is a* <where> type, meaning the associatee is exactly or a subtype of what the injection point expects, then a direct adapterless injection can be performed. Otherwise, the flowchart in Figure 2.3 is followed. We will go through each step of the figure to cover the details.

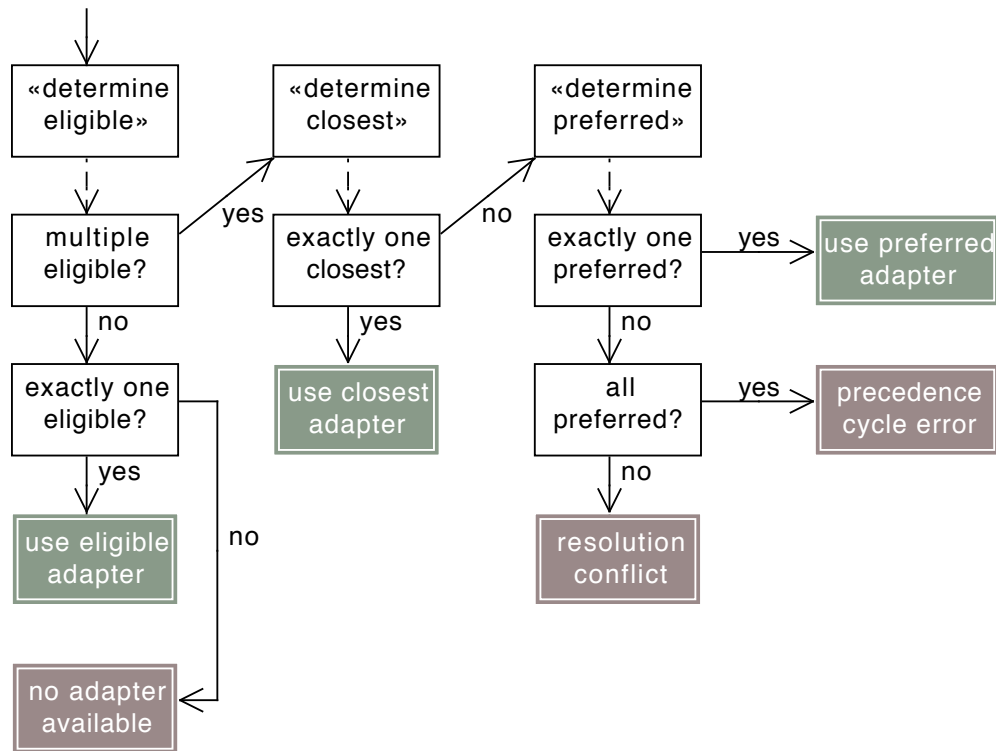


Figure 2.3: Flowchart of adapter resolution logic.

Eligible adapters

The steps in the first column are about finding *eligible* adapters. An adapter is eligible if it takes the associatee as its adaptee and if it is compatible with the injection point's type. In order to show what it means for an adapter to be eligible, we use the class hierarchies from Figure 2.4. The left side shows the hierarchy of possible injection point types and on the right side is the hierarchy of possible associatee types. In the middle are the adapters that connect the two hierarchies, denoted with dark boxes. The complete registry of adapters is A, B and C.

Now consider the association statement below, where we assume the type of the `httplogger` field is `HTTPLogger`:

```
associate field HTTP.httplogger with new AppLogger
```

In this case, only the B adapter is eligible, for it takes an `AppLogger` as its adaptee and is type compatible with `HTTPLogger`. Adapter A is not type compatible with

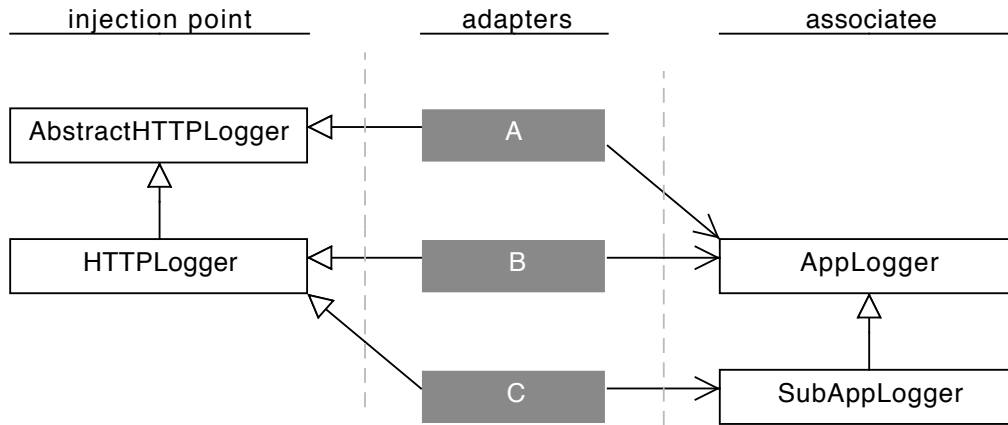


Figure 2.4: Example hierarchies for determining eligible adapters.

HTTPLogger and adapter C does not take an AppLogger as its adaptee, which renders both ineligible. Since only adapter B is eligible, the resolution is successful and finished. This means that if Gluer is running in checking mode, the association statement passes, and in runtime mode adapter B will be instantiated and injected.

However, if we have the association statement below, we get to a different situation. Note that both the injection point's type and the associatee's type have changed.¹ The types of both the injection point and the associatee are less restraining with respect to adaptation.

```
associate field HTTP.abstracthttplogger with new SubAppLogger
```

Now all three adapters are eligible, for they all take a SubAppLogger as their adaptee and they are all type compatible with AbstractHTTPLogger. In such a case, i.e. multiple adapters are eligible, more steps are required to find the most suitable adapter. The resolution mechanism will try to find the *closest* adapter. But before we discuss what that means, we consider the situation where *none* of the adapters are eligible.

Consider the slightly extended hierarchy in Figure 2.5, and the association statements below.

```
1 associate field HTTP.httplogger with new AbstractAppLogger
2 associate field HTTP.subhttplogger with new AppLogger
```

The adapter resolution will fail to find any eligible adapter for both associations. There is no adapter that takes the AbstractAppLogger as adaptee as declared on line 1, and there is no adapter that is compatible with the SubHTTPLogger class from line 2. If the resolution mechanism fails to find any eligible adapter, like in above two statements, the Gluer framework signals an error, whatever mode of execution it is in,

¹The names show their corresponding type, so in this case the injection point's type is AbstractHTTPLogger and the associatee's type is SubAppLogger

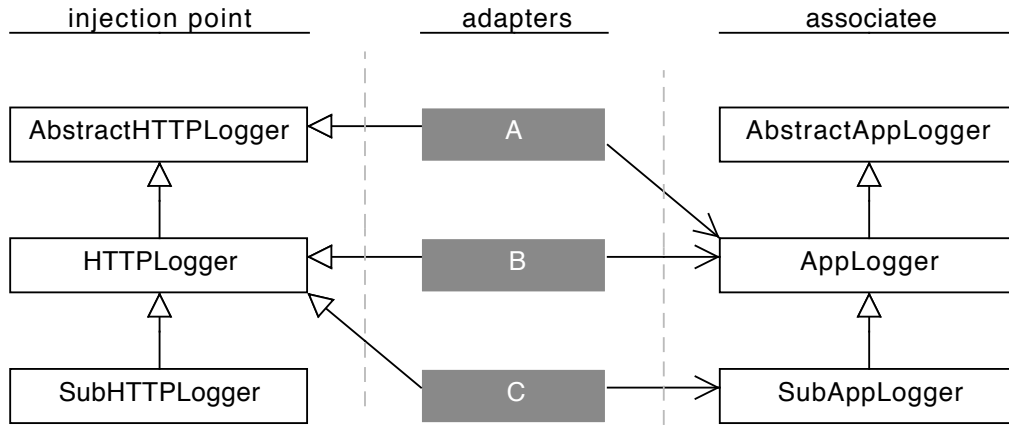


Figure 2.5: Extended hierarchies for determining eligible adapters.

i.e. the runtime mode or the checking mode.

Closest adapters

As shown above, it may very well happen that multiple adapters are eligible for a certain association. This means we arrive in the second column of the flowchart in Figure 2.3. Returning to the simpler hierarchy in Figure 2.4, consider again this association statement:

```
associate field HTTP.abstracthttplogger with new SubAppLogger
```

We had already established that adapters A, B and C are all eligible in this case. An important feature of our conceptual solution is that it finds the *most suitable* adapter. For Gluer this means that it tries to find the *closest* adapter.

We determine whether an adapter is closer than another adapter for a specific association by calculating *hierarchical path lengths*. Figure 2.6 shows a class hierarchy that helps to visualise this. The distance between two classes (or interfaces) is determined by the shortest path (counting the edges) between the two. Note that such paths only exist if one is an ancestor of the other. This way, the distance between classes X and Z is two, for the path via Y is the shortest path.

For an association, two such hierarchical path lengths can be calculated for each eligible adapter. The first is the distance between the associatee and the adaptee. The second is the distance between the adapter's type and the injection point's type. Returning to the association statement above, the two distances for the eligible adapters are shown in table 2.1.

For Gluer, the *associatee distance* is of more importance than the *injection point distance*. More distance between the injection point's type and the adapter's type simply means that the adapter may have more methods than expected, but those

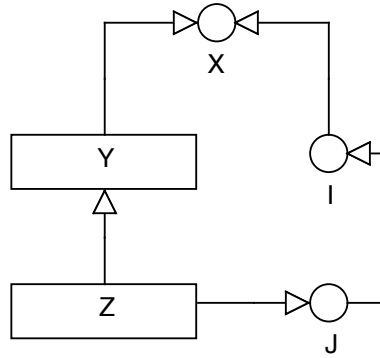


Figure 2.6: Path lengths in class hierarchies.

Adapter	Associatee distance	Injection point distance
A	1	0
B	1	1
C	0	1

Table 2.1: Distances for the adapters.

will never be called. Having a short distance between the associatee’s type and the adaptee’s type means the adapter can utilise more of the specialised “knowledge” it has on the object it is adapting, i.e. leveraging the fact that a larger set of methods is available. For this reason, Gluer first calculates the associatee distances of the eligible adapters and if that is not conclusive, i.e. still multiple options remain, it looks at the injection point distance. Considering table 2.1 again, this means adapter C is regarded as the closest for our example association. If however, adapter C would not exist, then adapter A is the closest. These simple rules are another example of how we have focussed on having transparent and understandable semantics.

Of course, above process is still inconclusive for the situation where two adapters have equal distances, and are regarded as the closest as well. We will see an example of this when *precedence relations* are discussed.

Adapter hierarchies

At this point we have discussed the greater part of how Gluer chooses which adapter is used for an injection. Since adapters in Gluer are plain Java classes, one is free to define entire hierarchical clusters of adapters. We can deduce from what we have seen so far, that these hierarchies of adapters need not be explicit, i.e. linked by inheritance. This means that one adapter does not need to extend another adapter per se, for the framework to pick the most specialised one. For example in Figure 2.4, adapter B takes an `AppLogger` as its adaptee, and adapter C takes a subtype of `AppLogger` as its adaptee. In this case, the hierarchy is implicit; C does not extend B, but the framework *will* choose C over B if it needs to inject a `SubAppLogger`. Of course,

one could still use explicit inheritance for adapters, just as one would with other Java classes.

Precedence relations

Consider the extended class hierarchy in Figure 2.7. Notice the newly added adapter called B2. This adapter takes exactly the same adaptee as the B adapter and it extends the same class as well. For Gluer adapters B and B2 are on equal ground. Consider the following association statement:

```
associate field HTTP.httplogger with new AppLogger
```

Now both adapters B and B2 are eligible (and no other), but also turn out to be equally *close*. Without any extra information the framework cannot make a choice and a *resolution conflict* would arise. But, we can supply this extra information.

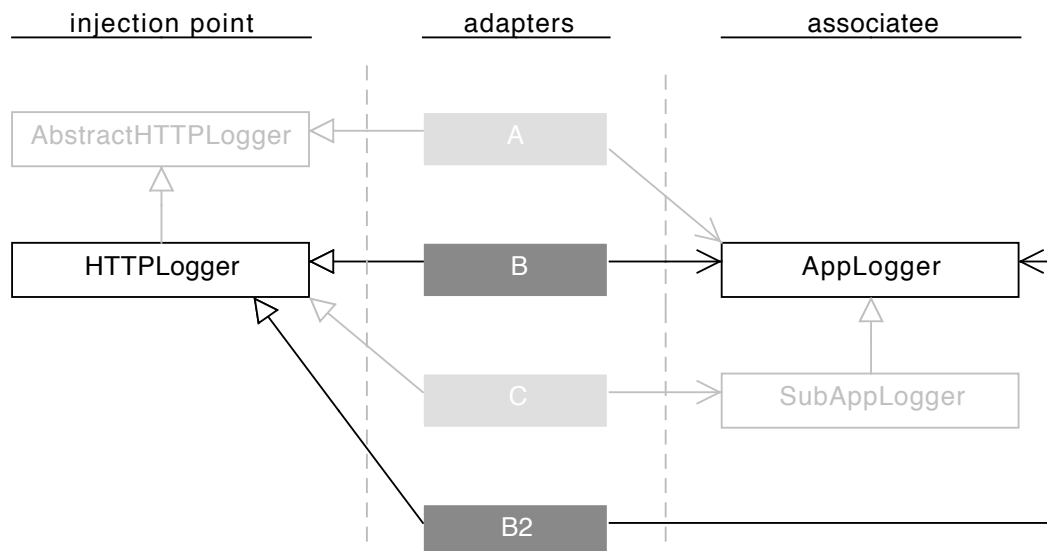


Figure 2.7: Extended hierarchies with equally close adapters.

We do this through the notion of *precedence declarations*. A precedence declaration declares which adapter should be used in case of equally close adapters. Precedence declarations can be defined in the same files as where the associations statements (can) go, and they have the following form:

```
declare precedence <comma-separated-adapter-class-list>
```

There must at least be two items in the precedence list. For any two items in this list, the one on the left has precedence over the one on the right. So if we want that adapter B2 is used in our example, we write the following:

```
declare precedence B2, B
```

Above statement is taken into consideration if and only if the adapters are equally close. In our example the adapters B and B2 are always equally close, but scenarios exist where two adapters are only sometimes equally close, depending on the association at hand. An example of when it would be dependent on the association statement, is when adapter B would also adapt to `SubAppLogger`, but adapter B2 would not. If the the associatee would be the `SubAppLogger` in this case, the adapters would not be equally close and the precedence relations are not considered, i.e. adapter B would be chosen, even though it is preceded by B2 in the above precedence declaration.

Since *all* classes tagged with the `@Adapter` annotation in the classpath are registered, precedence declarations are valuable in situations where one wants to vary which adapter to use for a particular project, deployment or development phase, without having to be explicit in which adapter to use (with `<using>` clauses, as will be explained in subsection “Overruling the adapter resolution”) and without having to remove adapters from the class-path. For instance, one could have an adapter suitable for unit testing and a similar adapter for use in production.

Precedence cycles

Having precedence declarations, one implicitly creates a directed graph of precedence relations between adapters. As will be shown, cycles in such a graph indicate precedence declarations that might rule each other out. The Gluer framework checks for these cycles when in *checking* mode and warns the developer about it if such cycles exist. This subsection explains how these cycles are created and how Gluer responds to them.

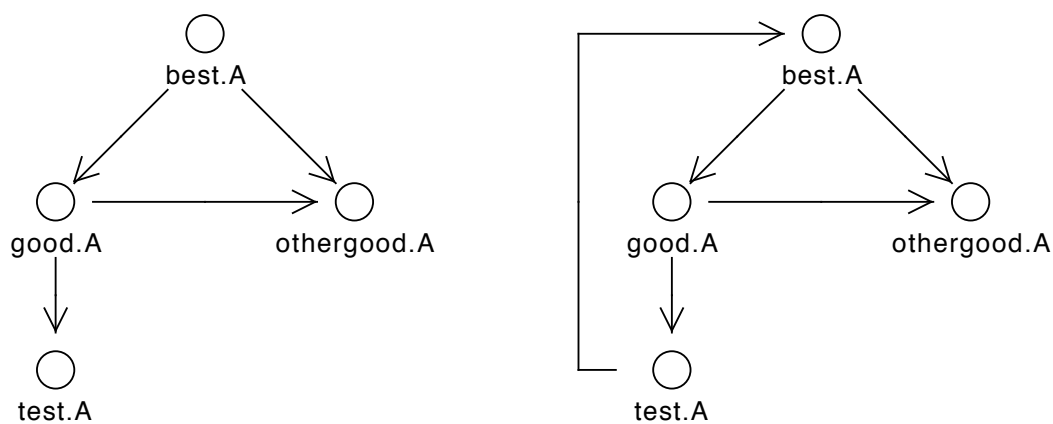


Figure 2.8: Left: correct precedence graph — Right: precedence graph with cycle.

Figure 2.8 shows two precedence graphs. A vertex in the graph is an adapter class, and the direction of the edge denotes what precedes over what.² In the figure, the graph on the left is created by the following statements:

²In the actual implementation, the direction is reversed. Every outgoing edge denotes that the current node is preceded by the node on the other end. The reason is that this results in cleaner and more efficient code.

```
declare precedence best.A, good.A, othergood.A
declare precedence good.A, test.A
```

This graph does not have cycles, so whatever combination of equally close adapters arises, the precedence declarations will not rule each other out. Note though, that the graph does not solve every resolution conflict that may arise for these adapters. If the set of equally close adapters only contains `othergood.A` and `test.A`, then the framework does not know which one to pick. Moreover, if the set only contains `best.A` and `test.A`, the framework cannot make a decision either, even though there is a path between those two adapter nodes. This is because separate precedence declarations are *not* transitive relations when looking at the entire graph. Another way of looking at it, is that only the so-called *induced subgraph* for the vertices at play is used for determining precedence.

The graph on right of the figure shows the same graph as on the left, but with the following statement added:

```
declare precedence test.A, best.A
```

As one can see, this creates a cycle in the graph. This does not necessarily pose a problem for correct runtime behaviour though, exactly because of the non-transitiveness as explained in the former paragraph. If the set of equally close adapters for a particular association contains only `best.A` and `test.A`, the runtime picks the latter. Moreover, if the set contains only `best.A` and `good.A`, or only `good.A` and `test.A`, no problems arise either. If, however, the set contains all three, that is `best.A`, `good.A` and `test.A`, the framework cannot make choice due to the circular relations.

Overruling the adapter resolution

The adapter resolution is always able to find a suitable adapter, if one exists and sufficient, non-cyclic precedence declarations are defined. Still, one may want to explicitly specify which adapter should be used for a particular association, i.e. overrule the adapter resolution algorithm. For this purpose the optional *using*-clause can be used.

```
associate <where> with <what> [using <adapter>]
```

The desired `<adapter>` can be specified, after the “*using*” keyword, but the entire clause is optional of course. The `<adapter>` should be a FQN. The Gluer tool will not go through the adapter resolution process if the adapter is specified, but it *will* check whether it is actually eligible for the association at hand.

2.2 Running the tool

The former section presented the association and precedence statements and how those are used. In this section we will see how these can be given as input to the Gluer tool and how to use the tool itself in general.

2.2.1 Input files

The association and precedence declaration statements are written in plain text files. For these files, we use the `.gluer` extension, but any extension will do. The statements can be mixed and the order of the statements does not matter. The statements can also be spread over multiple files.,

In order to increase portability of a composition using Gluer and to decrease the number of command-line arguments, a second file type is used: the configuration file. A single configuration file is supplied to the Gluer tool and its content consists of key-value pairs and comment lines. Every line that has at least one colon in it, is regarded as a key-value pair. More colons are considered part of the value. All other lines are ignored and serve as comments. The following keys are currently supported:

glue The value is a path to a `.gluer` file, relative from the location of the configuration file itself. This key can be specified multiple times.

verbose The value is either “true” or something else. In case it is `true`, verbose debug logging is emitted during the execution of Gluer.

An example configuration file is shown in listing 2.2. This configuration file instructs Gluer to load two `.gluer` files, and to omit any debug logging.

Listing 2.2: An example configuration file.

```
1 This is a configuration file. This line is considered comment.
2
3 glue: path/to/file.gluer
4 glue: path/to/other.gluer
5
6 Set the following to 'true', for debug logging.
7 verbose: false
```

2.2.2 Modes

As noted at the beginning of this chapter, the Gluer framework has two modes of execution. One is the *checking* mode, whereas the other is the *runtime* mode. Which mode is executed depends on how Gluer is started. Both modes receive a configuration file as discussed in the former subsection as input.

The checking mode

The Gluer framework is run in checking mode when it is started as a *normal* Java application, i.e. implicitly calling its `main` method. The following example runs the framework in checking mode:

```
java -cp .:gluer.jar gluer.core example.config
```

One sees that a class-path is given, including the Java Archive (JAR) file of the Gluer framework. The main class to execute, `gluer.core`, is also given as the application to run, and a configuration file is supplied as an argument.

The purpose of the checking mode is to see whether the associations, as found in all the `.gluer` files referenced by the supplied configuration file, are “correct”. Any warning or error found is reported to the user. What it means for an association to be correct is explained in Section 2.3.

As already briefly mentioned in the first chapter, all checks are performed statically. This means that no actual code that is referenced by the associations, precedence declarations or adapters is run. The classes are not even loaded into the JVM, so static initialisations are not executed either. The classes that are referenced by the associations, precedence declarations, and the adapters that one wants to have available, should be in the class-path though. It is in this class-path, supplied to the JVM, that the Gluer framework searches for adapters, but also searches for the referenced classes to gain typing information. This also implies that the adapter classes should already be in a compiled form, i.e. Gluer does not compile Java source code; that task is best left to the Java compiler itself.

The checking mode does not generate anything either. The only output it has are the warnings, errors, and possibly debugging messages if the “verbose” option is set, to the standard output.

The runtime mode

The Gluer framework is run in runtime mode when it is used as a Java *agent*. Section 2.4.4 explains how an agent works internally. In brief, a Java agent is started before the actual Java application and can influence class loading.

Influencing class loading is exactly what the Gluer framework does in runtime mode. Based on the association statements found in the `.gluer` files, the Gluer runtime changes classes before they are loaded, in order to insert instructions for the injections.

Starting an application using the Gluer framework runtime goes as follows:

```
java -cp . -javaagent:gluer.jar=example.config app.Main
```

As one can see, the same class-path is used (excluding the Gluer framework) and the same `gluer.jar` is used, but now for the `-javaagent` option. The configuration file is now an argument to the agent (denoted with the `=` sign). A new value, in comparison with the checking mode, is the class name that holds the `main` method of the application that is to be started.

In runtime mode, no checks are performed, other than syntactic checks in the `.gluer` files. A typical workflow therefore is to run the Gluer framework in checking mode first, and then use the same parameters (class-path, configuration, etc.) when executing in runtime mode. Of course one is free to change any parameter or statement and starting in runtime mode directly, skipping the checking mode. This dynamic workflow is possible because, as also stated in the section about the checking mode, the checking mode does not generate anything on which the runtime mode depends.

2.3 Checks

When the framework is used in checking mode, it tries to validate the association statements, based on the information it can retrieve statically. Note that it checks a lot, but it is not a full fledged software model checker with full coverage of every runtime path that may occur. The checks performed by Gluer are mostly about types, their compatibility and their behaviour in the previously discussed semantics. Here follows a descriptive list of the checks performed in checking mode.

Configuration

The first check is to see whether the supplied configuration file can be parsed. If not, this is reported and the framework immediately stops.

Adapter registry

The next step is to build an adapter registry. This registry contains all the available adapters, along with the types they adapt to and from. For each adapter, the following is checked:

- Check if the adapter class is declared public.
- If the adapter is not a top-level class, i.e. an inner class, check whether it is declared static.
- Check if the adapter is concrete, i.e. not an abstract class.
- Check if the adapter adapts from at least one type, i.e. it has at least one constructor that takes a single non-primitive argument.

If any of the above requirements is not met, it is considered an error. If such an error is found in the adapter registry, then the checking stops at this point.

Gluer files

Now knowing that the adapter registry is correct, we get to the meat of the checking process. First the `.gluer` files are parsed, and if any syntax error is found, this is reported and the checking stops. It may be the case that some files parsed successfully, while others did not. An earlier iteration of the framework would continue to check those files that parsed successfully. Since the addition of precedence declaration statements, we chose to stop checking altogether. If the checking process would continue, it could report errors that would otherwise be solved by the statements in the files that failed to parse, i.e. false positives, or it may not find errors that should be found, i.e. false negatives.

Precedence declarations

The next stage is checking the precedence declarations. For each precedence declaration, it is checked whether the specified class names are known in the adapter registry. If not, one of two errors is reported, telling either the class is not an adapter or the class cannot be found at all. If errors are found, the checking stops. If however no errors are found, the precedence relations graph is build and checked for cycles, as was discussed in Section 2.1.3. Cycles detected this way are reported as warnings.

Association statements

Now that the precedence relations graph and the adapter registry are both valid, the actual association statements can be checked. For each association statement, the following is checked:

The where selection clause Currently only one such clause is supported, the `field` clause. It checks whether the class exists, whether the fields exists and if it has the correct modifiers, i.e. not final and not static.

The what selection clause There are currently three such clauses. For all three it is checked whether the specified class exists. After this, it depends on the kind of the selection statement. In case of the `new` and `single` clauses, it is checked if the specified class has an (implicit) no-argument constructor. In case of the `retval` clause, it checks whether the arbitrary Java expression can be compiled.

The using clause If the association has a using-clause, it is checked whether the specified class is a known adapter. If not, one of two errors is reported stating

that either the class is not an adapter, i.e. lacks the `@Adapter` annotation, or the class cannot be found at all.

Overlap If no errors are found, it is checked whether the association does not have overlap conflicts with the other associations, like trying to inject into the same field.

If no error was found at this point, the types of the where and what selection statements are determined. If a using-clause was specified, it is checked that the specified adapter is actually eligible for this particular association. The eligible adapters are found based on the types of the where and what selections, as discussed in Section 2.1.3. If no using-clause was specified, it is checked if a single, most suitable adapter can be found for this particular association based on the normal adapter resolution algorithm.

This concludes the checking process. If no errors are found, then using the Gluer framework at runtime will not result in exceptions due to the framework itself (with the exception of bugs), given the following conditions:

- No changes to the parameters, i.e. the class-path, the classes in this class-path, the configuration and the statements in the `.gluer` files.
- No other process changes the classes at runtime.
- No new classes are introduced at runtime.
- All warnings have been resolved.
- None of the limitations have been challenged, such as ignoring the fact that injections can be overwritten in constructors or methods.³

2.4 Implementation and extensions

Now that the behaviour of our framework has been discussed, we discuss some implementation details behind it. The main reason we discuss it here is to show how extensibility has been implemented, as this was one of our goals. More details on the implementation that are not discussed here can be found in appendix C.

2.4.1 Implementation language: Clojure

The framework itself is implemented Clojure 1.4 [Hickey, 2008]. Clojure is a Lisp targeted at the JVM. The main reason for choosing Clojure is that Gluer is a lot about

³This is discussed in Chapter 3.

data and the processing of this data. The core of the Clojure language has a very powerful set of data structures and functions for efficiently processing and transforming this data. Extending the framework is easier as well, because of this focus on data and their surrounding functions, i.e. one does not have to deal with type systems, hierarchies and patterns. My preference as the programmer that implemented Gluer has had an influence on the choice as well.

2.4.2 Parsing

One of the design goals was to have the DSL grammar to be easily extendible, at compile time, but also at runtime. The runtime extendibility was a design goal to support a future plug-in system. The idea was that a plug-in would be able to add new statements or extend existing statements when loaded into the framework, and specify how those statements influence the static analysis and byte-code alterations.

To meet the design goal of such a dynamic parser, and since the one candidate Clojure parser we found that met this goal was not to our liking, a new parser has been developed for use in Gluer.⁴ The new parser is a so-called Parsing Expression Grammar (PEG) parser [Ford, 2004], and it has been developed as a separate library.⁵ More on the implementation details of the parser can be read in Appendix C.

Currently there is no plug-in system, although most of the fundamentals required for it are implemented. Still, the value of having a parser that is easily extendible is already noticeable when developing on Gluer. The grammar, i.e. the parsing rules, are defined in a simple Clojure-native data structure. Changing the contents of this data structure is all that is required to update the grammar, after which the framework can be rerun using the new grammar. There are no other steps to perform, such as running a parser generator and compiling the generated parsing code. This is one of the extensibility features Gluer has.

2.4.3 Multimethods

While Clojure is not an OOP language, it has a notion of types and has support for polymorphism and dispatching mechanisms. The most powerful dispatching mechanism offered by Clojure is the so-called “multimethod”. A multimethod is just a function on the outside. When defining a multimethod, one supplies the dispatch function yourself, which receives all the multimethod’s parameters. Together with the multimethod declaration, one defines an arbitrary number of concrete implementations

⁴The candidate parser is called Parsley (<https://github.com/cgrand/parsley>), but the resulting AST was not easy to use nor idiomatic Clojure. The new parser has been inspired by the “API” Parsley offers though.

⁵The library is called “Crustimoney” and can be found at <https://github.com/aroemers/crustimoney>.

of it, called methods. Each method specifies a dispatch value. When calling the multimethod, just as one would call a normal function (because it is a function), it evaluates the dispatch function, and subsequently it calls the method that has the result of the dispatch function specified as its dispatch value.⁶

The reason multimethods are discussed here, is because of how they are used in our implementation. We have already discussed how the Gluer parser makes the grammar easily extendible with new statements or parts thereof. The names of the rules in the grammar specification are directly used in the nodes of the Abstract Syntax Tree (AST) result of the parser. We leverage this fact by using multimethods that have a dispatch function based on those node's names.

As an example, let's consider the different types of *what* selection statements in the grammar. One of the things the framework needs to know, is the (static) type of the object the clause will return at runtime. Therefore, a multimethod called `type-of-what` is declared, which selects a concrete method based on the AST node's name. Currently there are three kinds of what selections, so three concrete methods have been implemented. Adding a new kind of what selection means adding another concrete method for `type-of-what`, amongst others. This approach is another example of how Gluer is designed to be extensible.

2.4.4 The Java agent

As said in Section 2.2.2, the Gluer framework has two modes of execution, and in the runtime mode the framework is run as a Java agent. This subsection explains what a Java agent is. A Java agent is a JAR file where one of its classes contains a `premain` method. The `MANIFEST.MF` file in the JAR specifies which class this is.

The `premain` method is called before the `main` method of the Java application is called, and has access to an `Instrumentation` object, from the `java.lang.instrument` package. Through this object, one can register `ClassFileTransformer` objects. These transformers get access to the Java byte-code of a class that is read, and may change the byte-code before it is actually loaded into the JVM.

These transformations are exactly what happens when Gluer is used as an Java agent. When a class is being loaded, it will go through our transformer, and will alter it if injections have been declared for that class. It is by using this approach, that we can have injections without touching any source code on the file system.

⁶Multimethods also take ad hoc hierarchies, another Clojure feature, in consideration. This is out of the scope for our purposes, but it makes powerful polymorphism and dispatching possible without the need for declaring types for the sake of dispatching.

2.4.5 Relation with Aspect-Oriented Programming

The way Gluer works at runtime has some relation with Aspect-Oriented Programming (AOP). The framework performs what in AOP is called *runtime weaving*. The *base* code is “weaved” with extra instructions. For AOP calling an *aspect* is the typical instruction that is weaved in and the weaving point, or the *join point*, is specified using *point-cut* expressions. Looking at Gluer from the viewpoint of AOP, the weaving point is selected by the `<where>` selection statement⁷ and the “aspect” that is called is performing the injection.

By making this relation clear, the interesting question arises if and how a framework like ours may benefit from an AOP platform. It may yield more powerful `<where>` and `<what>` selection clauses in the association statements, but the challenge here is that static analysis on those clauses must remain possible as well. We have chosen not to focus on this due to the added complexity and the lightweight nature of Gluer, but it is certainly an interesting follow-up research topic.

Also note that our conceptual solution has another common feature with AOP, namely *separation of concerns*. The concern of composition is separated from core logic the components offer. This makes the components more flexible in how they are deployed. The difference between our solution and AOP is how the weaving point is selected. With basic AOP, such as that what is offered by AspectJ, one selects code instructions of where to run an aspect, whereas in Gluer one can be oblivious where injection code is weaved. I.e., Gluer abstracts away from code instruction, as the selection is more about the structure of the classes. There are AOP implementations that abstract away from code instructions as well, such as the work by [Bockisch et al., 2011] on *Natural Aspects*, which has a join-point model based on higher level events. We think this kind of abstraction is important, for it fits our goal of having a simple solution.

⁷Although the statements that Gluer weaves into the base need not necessarily be at exactly that particular point.

Chapter 3

Validation and findings

This chapter contains the validation of our proof-of-concept and the conceptual solution it implements. It reflects on the requirements from Chapter 1 and discusses how we have validated the implementation semantics and what we have learned from Gluer. This chapter also serves as the foundation for discussing open issues and limitations in the next chapter.



3.1 Validating Gluer's requirements

The first thing we will cover is the validation of whether our concept is actually a solution to our problem. We will reflect on the requirements we have set in the first chapter. Especially [req 2] “keeping it simple” and [req 3] “worthwhile to use” are important. We have already seen that the technical [req 1] “injecting type-incompatible objects” is possible when we discussed Gluer in the former chapter, so we do not focus on that in this section.

We will validate the requirements for Gluer in two ways. First, we look at actual open source projects and reason about how Gluer compares to their composition approaches and if and how those projects could benefit from Gluer. More specifically, we look at projects that applied the Adapter pattern for integration purposes.

Next, we implement a single, more theoretical, use case three times. The *bare* implementation will not use any Dependency Injection (DI), the *Guice* implementation will use the Google Guice library for DI and the *Gluer* implementation will use our framework. As we will see, all three implementations do make use of the Adapter pattern. This more theoretical case allows us to apply metrics in the evaluation. This

way we can compare the effort required, in lines of code [req 3].

More importantly, for both validation strategies, we can compare the mental effort that the composition approach takes, i.e. how confident one can be on whether an implementation is going to work without problems and how much mental effort that took [req 2+3]. As discussed in Chapter 1, the more mental effort is spent on the foundation, the less one can focus on a more complex problem to solve.

3.1.1 Comparison with compositions in real-world projects

The first task was to find suitable real-world projects for the comparison. Our goal was to find Java projects that have applied the Adapter pattern for (unanticipated) integration of an independently developed component. For this, we asked Robert Dyer from Iowa State University if we could use their research tool called *Boa* [Dyer et al., 2012]. In brief, this tool offers a DSL to query an index of projects on SourceForge¹ and the revision data in the projects' Version Control Systems (VCSs). The search was performed by looking at commit messages that hinted at the introduction of the Adapter pattern for integration purposes. The query that was executed at the Iowa State University can be seen in listing 3.1.

Listing 3.1: The query script for Boa.

```
1 # Bockisch - detect adapter pattern usage using log messages
2 counts: output collection[string][int] of string;
3 p: Project = input;
4
5 when (i: some int; match(`^java$\`,
6     lowercase(p.programming_languages[i])))
7     when (j: each int; def(p.code_repositories[j]))
8         when (k: each int;
9             match(`adapter|integration|pluggable|binding|dependency
10                 injection|wrapper|legacy component|legacy
11                 system|legacy|component`,
12                 lowercase(p.code_repositories[j].revisions[k].log)))
13             counts[p.code_repositories[j].url][p.code_repositories[j]
14                 .revisions[k].id] <<
15                 strreplace(strreplace(p.code_repositories[j]
16                     .revisions[k].log, ``\r``, ``\\r``, true), ``\n``,
17                     ``\\n``, true);
```

The query selects Java projects (line 5) and retrieves all commit messages having one or more of the words “adapter”, “integration”, et cetera (line 7). Line 8, 9 and 10 produce the output, which looks like the following:

¹A website with open source software, at <http://sf.net>.

```
counts[http://zvtm.svn.sourceforge.net/svnroot/zvtm][4606] =  
    move nodes, but not yet moving every node component  
counts[http://zvtm.svn.sourceforge.net/svnroot/zvtm][4768] =  
    add stroke wrapper  
counts[http://zvtm.svn.sourceforge.net/svnroot/zvtm][629] =  
    added L(3) lenses, cleaned up key bindings  
counts[http://zvtm.svn.sourceforge.net/svnroot/zvtm][636] =  
    more lenses, changed bindings for misc cmds  
counts[http://zwordtools.svn.sourceforge.net/svnroot/zwordtools]  
[116] = V3 draft: keys handled using hooks rather than  
bindings
```

The query results show the URL, revision number² and the matched commit message. The query resulted in 52712 of such lines in total. The selection process was by manually scanning all the commit messages for candidates. The size of the projects belonging to the commit messages also played a role, as very large projects make it relatively difficult to reason about. To make this selection process easier, a Python script was created to group the commit messages per search term and present them in HTML documents (see Appendix A). The candidates we found by scanning the results were inspected for usefulness for doing an evaluation.

The Argo project

What we found is that most projects use the Adapter pattern as part of their initial design. An example project that did *not* have the Adapter pattern initially is *Argo*³, a JSON parsing library. At revision 20 the developers added an adapter for integrating it with a SAX parser and a JDOM parser. Looking at the source code changes in Argo when these adapters are introduced, we see that it was a manual process: the client of the library has to initialise the adapter itself and pass it to the actual parser. To make this possible, the source code of parser itself needed changes as well. After this change though, using another adapter is now quite trivial, since the way the client has to supply the adapter is a nice example of basic Dependency Injection.

The reason these adapters were added to Argo is somewhat ironic though. They have been added so that the library can be integrated in applications that use a SAX or JDOM parser, but want to read a JSON document. So, the adapters were not really added to *use* the SAX and JDOM parsers, but to provide them as *integration* points. In summary, Argo uses adapters to integrate integration points. This is an example where the developers anticipated possible uses of the library and made it extendible. So, while the adapters have been added later in the project, it was not because of

²The repositories use SVN as their VCS.

³<http://sourceforge.net/projects/argo/>

unanticipated integration of a separately developed component.

The TimeLogNG project

As a matter of fact, we have not found a project that did have this property. We did find a nice example of a more general approach in using adapters, namely the Eclipse Framework. Where the Argo project has a specific interface for adapters to implement, the Eclipse Framework has a more general approach in using adapters [Beaton, 2008], that has similarities with Gluer. Just like Gluer, it has an adapter pool. The class `AdapterManager` has a method to register adapter *factories*, together with the interfaces the adapters it creates have and what types it takes as adaptees. This registration of adapter factories can also be done declaratively in an XML. A second method in `AdapterManager` can be used to retrieve a suitable adapter factory, based on an object and the expected interface.

The project *TimeLog Next Generation*⁴ was also found using Boa. It is a tool used to track time spent on different tasks, and it is implemented on top of the Eclipse Framework. It uses the `AdapterManager` in order to integrate the TimeLogNG data model classes into Eclipse, using adapters that let Eclipse know how to display the model classes. Listing 3.2 shows how the registration is done programmatically.

Listing 3.2: TimeLog registering its adapters.

```
1 public void init() {
2     delegate.registerAdapters(new
3         SimpleWorkbenchAdapterFactory(
4             new ClientAdapter()), Client.class);
5     delegate.registerAdapters(new
6         SimpleWorkbenchAdapterFactory(
7             new ProjectAdapter()), Project.class);
8     delegate.registerAdapters(new
9         SimpleWorkbenchAdapterFactory(
10            new TaskAdapter()), Task.class);
11    delegate.registerAdapters(new
12        SimpleWorkbenchAdapterFactory(
13            new DefaultTreeSetAdapter()),
14        TreeSet.class);
15    delegate.registerAdapters(new
16        SimpleWorkbenchAdapterFactory(
17            new DefaultListAdapter()), List.class);
18    delegate.registerAdapters(new
19        SimpleWorkbenchAdapterFactory(
20            new PeriodAdapter()), Period.class);
21 }
```

⁴<http://sourceforge.net/projects/timelogng/>

The `init` method in the listing is called when the Eclipse Framework is started. In it, one sees that a `delegate` is called to register adapter factories, for each type of model class (e.g. `Client` and `Project`). The `delegate` is actually the `AdapterManager` from Eclipse, whereas the factory (i.e. `SimpleWorkbenchAdapterFactory`) and the adapters are from TimeLogNG itself.

Comparison with Gluer

Now how do these projects compare to our framework? Let's first look at the Argo project. A potential downside of their approach is that they needed to update the source code in order to integrate their adapters. We say potential, because our framework, in its current form, would not have been able to integrate the adapters. Our framework only supports injection into fields, and that would not have been enough. We noticed this limitation more often when looking at the projects in the search results of Boa.

For the Argo project however, we still can make a case for Gluer. As said before, the adapters were added to the project in order to supply integration points for certain use cases. It is good that they did this, including the way it supports Inversion of Control (IoC) and preserves statically checking on whether a suitable adapter is used (although this leads to a tight coupling). But as we stated before, it is difficult or impossible to foresee all use cases. From this point of view, we think the Argo project would still benefit from our framework. Namely, by using Gluer, intertwining the core parsing logic with the integration logic can be prevented. One would make Gluer responsible for the concern of integration and let the library focus solely on parsing. One would write an adapter around the library API for integrating Argo in an application not developed with Argo in mind. Another advantage of using Gluer, is that adding more intergrations or extensions would use the same generic framework, instead of multiple, specialised approaches and removes the need for updating the source code again, possibly complecting the core parsing logic more.

The Adapter pattern as used by the Eclipse Framework is already more generic. The adapter pool it manages is very similar to the one in Gluer. Their approach shows how well the Adapter pattern is able to decouple components, while still having a tight integration. However, registering and using the adapters is done procedurally, as was evident in listing 3.2. This means that if no suitable adapter is available at some point, this is only discovered at runtime. Since Gluer is also a DI framework and is declarative instead of procedural, it finds such issues before the application is run. This has a huge advantage over the approach Eclipse and shows the strenght of our solution. Moreover, our framework would also remove all of the intertwined boilerplate code, such as registering the adapters and retrieving an adapted object.

Still, the current implementation of our solution would not be sufficient. In the case of how Eclipse uses the Adapter pattern, more types of injections would be needed

as well as support for injecting collections.⁵ A more interesting problem however, is that the Eclipse Framework uses several adapters for a single object; each for adding a specific chunk of behaviour to an object. This is a good example of composable behaviour. Doing this with our framework however, would mean that we would need to inject an adapter with the same adaptee instance multiple times, once for each interface Eclipse expects. This would make for some tedious and complex code, undoing the original purpose of having composable, decoupled behaviour on a single object. It is because Eclipse explicitly asks for a suitable adaptation of an object on runtime, based on a certain interface, why this works well in Eclipse. Gluer does not have this explicit and procedural nature of retrieving adapters, as Gluer is declarative and is designed to be applicable to code that was not implemented in this way. It is a question whether Gluer should support this “dynamic” behaviour, i.e. being able to ask Gluer for an adaptation of an object through an API, in order to support the composition mechanism as found in Eclipse. Note that this is a problem due to the type system of Java, as the way Eclipse uses adapters for composable behaviour is actually a way of dealing with the fact that Java does not support *traits*.

3.1.2 Theoretical use case implementation

To have a more concrete understanding of how Gluer compares to other common approaches in integration compositions, we have implemented an application according to our example we have used throughout the thesis. It consists of a simple application, using a logger and an (independently developed) HTTP component that uses its own logger. This is not the preferred situation. It would be better if the HTTP component uses the logger of the application. Both situations were already depicted in Figure 1.1 in Chapter 1.

We have implemented this use case three times. The first implementation is a simple base case, not using any DI framework, hard-coding every dependency. The base case is first implemented with the bad situation, i.e. as if the application and the HTTP component were developed separately, and then it is changed to get to the preferred integrated situation. It are those changes that are interesting and that we have evaluated. Next, we have altered the base case so it uses Google Guice as its DI framework. We have evaluated the changes to get to the integrated situation again, and compared this to the simple, hard-coded implementation. We have done this a third time on the base case, but then using our Gluer framework. The source code and the explanation of the three cases can be found in Appendix B. The subsections below discuss our findings.

⁵This is discussed in Chapter 4.

Hard-coded implementation

As can be read in the appendix, the source code needed changing to get to the integrated situation. While those changes were small and easy, it did involve opening up both components. In our simple use case this is feasible, but when dealing with more complicated components, it might not be that easy. These results are not unexpected for this base case.

Google Guice implementation

To incorporate Google Guice, some impactful structural changes needed to be performed, as can be seen in the appendix. We think that needing to change the structure of classes, in order to incorporate the Guice framework, is a downside.

In Guice, the advertised way of specifying what to inject where, is by writing so-called *module* classes. One basically tells Guice which type to bind to another type or instance. The `@Inject` annotation tells Guice to inject something at that place, and Guice then injects that what is bound to the type of that place by looking in the registered binding modules.

Reflecting on the Google Guice approach, we can say that it is fairly simple to get to the desired situation. Less changes were needed compared with the hard-coded implementation. While this is a great improvement, there are some downsides to this approach as well:

- Even though the `@Inject` makes it clear that something is injected at a certain place, one does not know for sure whether the injection will actually take place. If an incorrect or incomplete module is registered, one does not know until it fails at runtime.
- Debugging might be more difficult, since object instantiation goes via the Google Guice reflection “magic”.
- As already mentioned, Using Google Guice imposes certain structural restriction on the source code. For example, a field is injected *after* the object is fully instantiated, so the field cannot be used in the constructor. We will see that *when* to inject into a field is a trade-off, as we evaluate the Gluer implementation, but for Guice we think injecting before the constructor is executed would be better. This would have prevented some of the necessary changes. Another structural issue, is that when instantiating an object that instantiates another object that requires injections, those injections are not performed automatically. That is to say, the dependency injections are not recursive. This imposed even more changes to the base case.
- While changing what is injected somewhere was easy, it required changes to

the source *and* the developer of the HTTP component needed to anticipate in advance that one might want to inject another logger. This means that even when Google Guice is used, integrating separately developed components may not be as simple or even possible as we would like to see with our goals.

All in all, Google Guice is still fairly intrusive. That, and the fact that compile-time checked injections are not available, makes the Guice approach not fulfill our goals.

Gluer implementation

Now that two cases have been covered, we will describe our findings on applying our Gluer framework. For this, we took the base case again. This time it was our intention not to touch any existing source code, while still getting to the desired integration. Having just a `.gluer` file in place proved not to be enough to get to the desired situation, though. We had to change the `HTTP` class a little, as can be read in the appendix. With this change however, the integration works as desired.

The change had to do with the choice of whether to inject a field before the constructor is called, or after. This is actually a hard choice, and it depends on the case. Maybe a future addition for Gluer is to be able to specify the time of field injection. This is discussed in more detail in Chapter 4.

A positive point is that the injection/association can be checked, by using the *checking* mode of Gluer. However, it only checks that what is declared in the `.gluer` file. Since we do not want to change any source code, the Gluer framework can by design not check whether an association is missing. In case of Google Guice, this would be an option (and it is, at runtime), since injection points are annotated.

Both the Guice and Gluer approach require good knowledge of the components at hand. Of course, this is even more the case in the hard-coded implementation. In case of Gluer, the injection declarations are centralised, correlate *what* is injected nicely with *where* it is injected, and are easy to understand. In case of the Guice approach, the injections (both *what* and *where*) are scattered throughout the source code, but this does make the injection points more explicit. It is a matter of opinion what is preferred; centralised specifications, but sort of magic in the source, or scattered though more explicit/less magic. A specialised editor for Gluer that shows where injections will take place, may help in this regard. Such editors are also seen with AOP, which show where a pointcut is influencing the execution flow. Again, it is a matter of opinion whether this is desired. However, in light of *our* goals, the preferred way is the centralised way, as it is for Gluer. This is because this way no source code needs to be changed, and more importantly, it can be “tagged” on later to support extensions and integration use cases that had not been foreseen.

One last minor downside for the Gluer approach might be that running the application is a little more complex, having to use an agent. In case of Guice, one simply

needs to have the framework libraries in the classpath. Still, the power Gluer gives is making up for this downside, as the process for getting to the desired situation was simple and effective.

3.1.3 Overall requirements validation

Now that we have seen how Gluer compares to other approaches, we can evaluate whether the requirements and the goals we had set are met. First we consider each conceptual requirement from Section 1.2.

Requirement 1: Capable of using type-incompatible objects

“[A solution should] have capabilities for objects to use other, possibly type incompatible, objects, without needing to change any source code. The solution’s scope is for one-to-one relations between objects, i.e. the new, incompatible dependency should have the same semantic interface as the compatible (legacy) dependency.”

Clearly this requirement is met, as it is the central goal of Gluer. It is also seen in action in the theoretical use case implementation.

Requirement 2: Keep it simple

“[A solution should] keep it simple. The solution should be simple, intuitive and easy to fathom from a user’s perspective, i.e. it must be possible to quickly have a complete mental model in one’s head of what is happening under the hood. Also, the steps to perform to incorporate the solution should be kept to a minimal and simple.”

On one hand, we have tried to keep Gluer simple, both externally as internally. We think we have succeeded in doing that. The external input that Gluer uses (the `.gluer` files) are easy to read and the semantics are well-defined and intuitive. The way the execution modes are invoked require no new knowledge or tools; it is part of the bare Java ecosystem. Internally we implemented the framework such that extending it should not be too difficult, as we have already given examples of in Section 2.4, if one does not deviate from the core concept that Gluer implements.

On the other hand, a critical note needs to be placed here. We have seen some limitations of Gluer already, such as not overwriting hard-coded field initialisation. Also the selection clauses in the association statements are the bare minimal. We will see some more current limitations when we discuss future improvements in Section 4.2, and for some of those improvements, a big challenge is to keep Gluer simple. For

example, the possibility to select an already existing object as the associatee (such as the object in the `App.logger` field in our theoretical use case implementation.) would be a great addition, but may also complicate the DSL quite a bit.

Requirement 3: Worthwhile to use

“[A solution should] take a considerable amount of work out of the developers hand, i.e. it should be worthwhile to use the solution. The work required can be quantified in for example lines of code that need to be written or the amount of testing required to be reach a certail level of confidence about the robustness of the composition.”

Even though the Gluer framework now only has the bare essentials regarding supported injection points and associatee selection, we can already say that a framework like our may be very worthwhile. If we consider our theoretical use case implementations again, the ease in which we integrated two separate components really stands out. Even more, with the Gluer implementation we were confident that the injection will take place, something that was not the case with the Guice implementation, even though that framework has seen a lot more development. What we mean by that, is that our approach can become even more powerful and worthwhile to use, since it is only in its infancy.

Next up is the evaluation of each goal we had specifically set for an implementation of our conceptual solution, as described in Section 1.2.2.

Goal 1: Check for errors before run-time

“An implementation should be able to check for errors that may arise from using the solution. The more pre-runtime checks within this scope, the more developer can be confident that no issues are encountered when the application is run. Note that this is not about errors that lie beyond that of what the solution adds to the composite application.”

The goal to check for errors “*that may arise*” is difficult to quantify. We did however implement many checks we can do based on static analysis on types. We have added some supplementary checks, such as checking whether the adapters follow the specific Gluer rules and whether the arbitrary Java code for the `retval` selection expression compiles.

That said, Gluer did not intend to be a full-fledged code analyser. This means, as also noted in the use case implementation, it cannot detect whether a field never gets its dependency injected, in case the developer had forgotten to specify it in the `.gluer` files. Still, for the injections the developer *does* specify, one can be rather

confident that no issues arise due to what the Gluer framework performs, before running the actual application.

Goal 2: Automate choice of adapter

“An implementation should automate the choice of which adapter to use as much as possible, with clear semantics. Such automation could be accompanied by ways of influencing this automation.”

This fairly technical goal has been met in Gluer. Based on solely the types of an injection point and an associatee, we consider all eligible adapters and try to bring it down to the most suitable adapter. This will work in most cases, and otherwise we have added two ways for the developer to assist the framework in making a choice; either by precedence rules or by appending the `using` clause to an association statement.

Goal 3: Extensible with new selection clauses

“For association statements, many choices can be made. For example, what kind of injection points it supports and how one selects these points. Therefore, the framework should be simple to extend with new selection constructs in association statements.”

As explained in Section 2.4.3, the implementation parts that deal with the analysis, checking and execution of the selection expressions from our DSL, are implemented using multimethods. This, in combination with the compile- *and* runtime alterable parser grammar, makes the Gluer framework fairly extensible in this regard.

3.2 Adapter selection semantics validation

The most important aspect of the semantics in Gluer is the adapter resolution for type-incompatible associations. To make sure this works the way we intended, we have implemented a test suite that tests all possible cases the adapter resolution mechanism can be confronted with.

A class hierarchy, or actually two, that covers all possible situations is shown in Figure 3.1. Testing the permutations of all possible associations one could write for the class hierarchies, together with what the expected outcome is considering the adapter resolution semantics, validates our implementation.

Listing B.18 implements those tests and expected values. In this test function, the actual implementation function called `get-adapter-for` is used (as seen on line 5). That function uses a function called `leveled-supertypes-of`, which normally

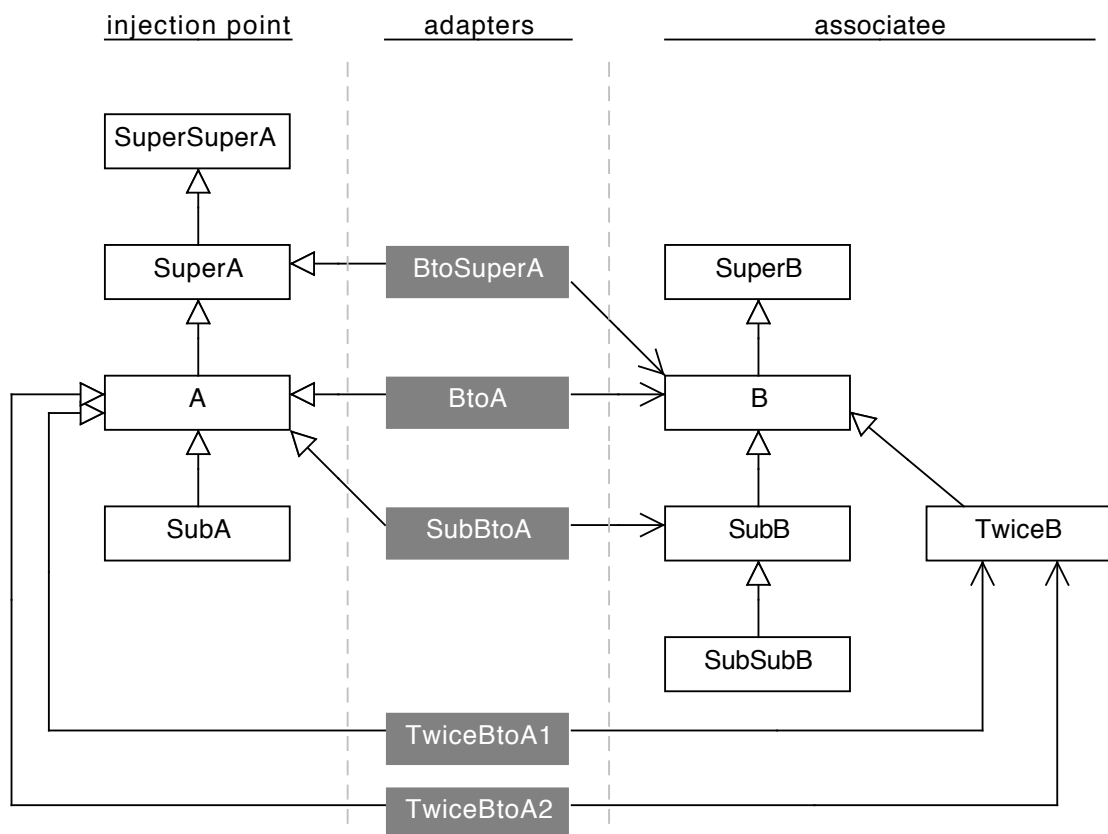


Figure 3.1: Test class hierarchy.

uses class definitions residing on the class-path for determining the supertypes of a class. For our test function, it is redefined on line 2 to use our own datamodel that represents the class hierarchy from the figure.

On line 6 to 14 the test cases are described that should succeed in finding a suitable adapter. One sees three strings on each of those lines (ignoring the lines with `; comments`). The first string is where to adapt from, i.e, the adaptee or associatee. The second string is where to adapt to, i.e, the injection point's type. The third string is the name of the adapter that is expected to be returned from the `get-adapter-for` function. Any deviation from that would fail the test function.

Line 17 to 24 tests the cases where no adapter should be found. In this case, the `get-adapter-for` function should return an error. The test on line 28 is similar, but in this case a warning is expected from the function. Again, any deviation from the expected error or warning fails the overall test.

When the test function is run, it succeeds, thereby validating our implementation of the adapter resolution as described in Section 2.1.3.

Chapter 4

Discussion

This chapter wraps things up, by discussing the current open issues, possible improvements, new open questions that have arisen and related work. At the end of the chapter a summarising conclusion is given.



The former chapters discussed most parts of our research, starting from a problem statement. The very general problem, that of improving composition and thereby reusability of software components is what we started with. This thesis is about improving a more specific problem from this area.

We identified that in OOP more often than not, a class cannot be used by another class, when those have been developed separately. Also, in many cases the source of a class needs to be changed in order to let it use another class then it currently does. In other words, classes are most of the time not programmed in a way that they offer extension, or rewiring of its dependencies. Even if they have been programmed that way, most of the time they cannot be used because of type incompatibility.

As this thesis has described, we have found a powerful yet simple solution to this. Our solution is having classes use other classes than they already do, in a way that type incompatibilities are less of a problem, and no existing source code needs to be touched.

Of course the solution and our implementation are not without limitations. Also new issues have arisen. This chapter will give an overview of those limitations and open issues. Related work will also be discussed and a final conclusion will be given.

4.1 Open issues and limitations

The former chapter identified some issues with our framework, which are discussed in more detail in this section.

4.1.1 Expressiveness of DSL

The first subject we want to touch upon is the limited expressiveness of the currently implemented selection parts of the association statements. Both the selection of the injection point and the selection of the associatee are currently not powerful enough to glue together software components in many scenarios. As already explained in Chapter 2, we kept the expressiveness limited on purpose, as it was not our main objective with our prototype. Nonetheless, the currently implemented selections are not sufficient in many cases. Consider for example the following scenario.

```
1 public class A {  
2  
3     private B b;  
4  
5     public A(Config c) {  
6         b = BFactory.create(c);  
7     }  
8 }
```

In class `A` above, the actual implementation of `B` that will be used is not hard-coded in the class. One can also see that the returned `B` instance from the factory might depend on a configuration. Two issues arise when one wants to inject another object in the `b` field. The first is that the injected value would get overwritten by the statement on line 6. This issue is discussed further on in this section, so let us focus on another issue. The `Config` class might contain valuable information about what kind of object to create in the factory. If we want to inject an associatee in the `b` field using Gluer, we might need that information for our associatee selection as well. Or more general, one might want to have more context. Context that is available at runtime. So, to gain more expressiveness in these scenarios, the associatee selection should be able to capture variables.

Of course, one could argue about what context should be possible to capture. The context that could be made available for the associatee selection is dependant on the injection point that has been selected. As already mentioned in Section 2.4.5, the relation of Gluer and AOP is strong, but has an important difference. Our injection point selection is place oriented, not execution/instruction oriented. We think this is a nice property of our framework, as it is simpler for a user to reason about. But a

side-effect is, is that it is not obvious to the developer using Gluer what context is available. Take for instance class `A` from above listing. We know that in the current implementation the `c` argument is available during a field injection. But this is not clear to the user of Gluer, nor should it be, and the implementation could change in the future. It will be a challenge to find a nice way to use context of the injection point in the association statements, without losing the place-oriented property of the injection point selection, though a worthwhile one since it makes Gluer suitable for more composition scenarios.

While we have talked about context in the sense of what is available in the running program, another context is that of Gluer itself. For instance, a nice addition to the `retval` associatee selection would be that the class name of where the injection takes place would be available to the code that is specified as `retval`'s parameter.

One can also think of more expressiveness by adding more places to select, such as parameter injection or return value injection. Adding these would make Gluer more versatile.

4.1.2 Concurrent use with other injection frameworks

Another limitation, which is likely common with other frameworks, is that Gluer will not always play nice with other injection frameworks. Thus, an application that uses Spring or Google Guice, is less suitable for use with Gluer. Also, application that are run in a container, such as application servers, use their own specialised classloaders and influencing the starting of the application using an Java Agent might be problematic.

4.1.3 Final classes as the type of the injection point

Due to the static type system of Java, with no support for duck typing, structural typing or any dynamic concept alike, a technical limitation arises when the type of a desired injection point is declared final. One cannot define an adapter for this type, as it cannot extend the final type.

4.1.4 When to inject into a field

An issue that was already touched upon at the start of this section, and which was also evident in the validation of Gluer in Chapter 3, is the “time” at which a field injection takes place. The field injection takes place when an object is instantiated. But, a choice has to be made when exactly this injection takes place, i.e. *before* the constructor is run, or *after*. There is a trade-off to be made. The advantage of injecting the field before the constructor is run, is that the code in the constructor can directly use that what is referenced by the field, but the downside is the risk that the injected associatee

will be overwritten by the instructions in the constructor. One could also choose to inject the field after the constructor is run, so the field will not be overwritten, but this has the downside that the constructor cannot use that what is referenced. Of course, an injected object can always be overwritten in another method, so the overwriting problem is more general.

Currently, Gluer injects into a field *before* the constructor is run, in order to have the advantage of using the associatee in the constructor. Related to the expressiveness issue discussed at the beginning of this section, an idea would be to add possibilities to influence the time when an injection takes place. Another possibility would be to add means to “ignore” statements in the class that would otherwise overwrite the injected field. Of course, one should be careful to keep the place-oriented property of the Gluer DSL, as was described in the subsection about expressiveness.

4.2 Future improvements

Along the way of implementing and reasoning about our concept, we have thought of possible ways of improving Gluer in the future. This section gives a quick overview.

4.2.1 External improvements

External improvements are those that are visible to the user, e.g. the DSL or the command-line invocation.

Context variables for `retval` The `retval` associatee selection may benefit from having context available, as was already described briefly when discussing the expressiveness of the DSL in the former section. One can think of context information such as the injection point for that particular association statement.

Arbitrary arguments for `new` and `single` Currently, the `new` and `single` selection statements take only a class name and expect the class to have a no-argument constructor. This could be extended in a way that argument expressions can be supplied. This would have eliminated the need for a factory in the Gluer implementation of our theoretical validation test case (section 3.1.2). Possibly this extension might also benefit from the former item, i.e. context variables.

Inject into static fields Other injection points than an instance field have been discussed already, such as injecting into parameters of methods. Another option is to be able to inject into *static* fields. Of course this has the same *overwriting* issue as with injection into instance fields that one has to consider with this extension.

Make configuration files optional It is slightly overkill to have a configuration file if only a single `.gluer` file is used for a particular composition. The Gluer framework could be made to support both a configuration file and an associations file as input.

Warn on incomplete override An extra static analysis check could be added, which checks whether an adapter that *extends* a class reimplements every method that class defines, and emits a warning if it does not. A choice has to be made here when implementing this though; whether the check also incorporates the methods that the superclass of the extended class defines, or even deeper.

4.2.2 Internal improvements

Internal improvements are those that are not visible to the user, but are about the implementation of the framework itself.

Support for generics Currently, the Gluer framework does not support generics. More precisely, the generics are ignored by the tool when figuring out the type of for instance a field. A future improvement would be to add support for generics. This also opens up the possibility to add support for injection of collections.

Logic programming The implementation of the adapter resolution algorithm might benefit from being reimplemented at a higher abstraction level, such as Logic Programming (LP). For Clojure one could use *core.logic* or *Datalog*, where the latter is simpler and probably also sufficient. The advantage of reimplementing it using LP is that it will become clearer how it works internally, which makes it easier to maintain or change its behaviour. Above libraries may even be more complicated than necessary, and using the standard library as described by [Engelberg, 2013], "*Logic Programming is Overrated*", might be sufficient.

4.3 Related work

The paper from Mezini et al. discusses so-called *Composite Adapters* [Mezini et al., 2000]. Composite Adapters are a group of concrete adapters that work together. Just as Gluer is (by default) implicit in which adapter it uses for an association, composite adapters define a scope around concrete adapters, where type *lifting* and *lowering* the return values from calls to the adaptees happen implicitly. E.g. if adapters X and Y are within the same composite scope, adapter X is automatically used whenever adapter Y tries to use an incompatible type that adapter X can adapt to. Mezini et al. propose to extend OOP languages with constructs that make such composite adapters available to source code tries to use incompatible types, which are then implicitly adapted using the available composite adapters. Among these proposed constructs, are keywords that

explicitly lift or lower objects to another type, whenever multiple adapters can resolve a type incompatibility.

There are two major differences between the approach of Mezini et al. and our approach. The first and most important one is that the Composite Adapters approach imposes its use upon the “client” code, i.e. it is intrusive. This yields them the benefit of having the entire language at hand for selecting which object the client code uses, without introducing a new language. The downside however, is that their approach cannot be used for legacy code without changing it. Enabling just that was one of our goals, which Gluer provides.

The second difference is that Gluer has the notion of implicitly using the *most suitable* adapter available, in order to avoid resolution conflicts as much as possible. The work of Mezini et al. does not have such a mechanism in place. Then again, since their work has scopes for adapters, resolution conflicts probably occur less often. Adding the concept of scopes to the adapter registry of Gluer might be a worthwhile topic for further research.

4.4 Conclusion

As type incompatibilities between separately developed components are common in Object-Oriented Programming and changing legacy source code is not always an option, composing such separately developed components is a non-trivial task. Patterns and frameworks exist to overcome this issue, but most of these solutions:

1. impose a certain structure on the source code,
2. are not compatible with each other, and
3. require the developer to foresee future use cases of its component in order for the pattern or framework to be advantageous.

Though the first two issues are not insurmountable, it is impossible to foresee every future use case. Trying to compose components in an unforeseen way remains difficult, even with the pattern or framework in place.

With our research and proof-of-concept implementation we have shown that by combining the Adapter pattern and Dependency Injection in a certain way, type incompatibilities can be overcome, without the need to change any source code. I.e. the solution we have described and validated is non-intrusive, and thereby mitigates the third issue altogether. As is discussed in this chapter and the former chapter, the first two issues are not completely solved, though for the first issue this is a matter of extending our proof-of-concept.

We have also shown that by having clear semantics and a declarative Domain-Specific Language that our solution can be powerful yet simple. Our proof-of-concept has a strong static analysis, increasing the benefits of adopting our approach.

Bibliography

- [Beaton, 2008] Beaton, W. (2008). Eclipse Corner Article: Adapters. <https://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html>.
- [Bockisch et al., 2011] Bockisch, C., Malakuti, S., Akşit, M. and Katz, S. (2011). Making aspects natural: events and composition. p. 285, ACM Press.
- [Dyer et al., 2012] Dyer, R., Nguyen, H., Rajan, H. and Nguyen, T. (2012). Boa: analyzing ultra-large-scale code corpus. p. 87, ACM Press.
- [Engelberg, 2013] Engelberg, M. (2013). Logic Programming is Overrated. <http://programming-puzzler.blogspot.nl/2013/03/logic-programming-is-overrated.html>.
- [Ford, 2004] Ford, B. (2004). Parsing expression grammars. pp. 111–122, ACM Press.
- [Fowler, 2004] Fowler, M. (2004). Inversion of Control Containers and the Dependency Injection pattern. <http://martinfowler.com/articles/injection.html>.
- [Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. and Vlissidis, J. (1995). Design patterns : elements of reusable object-oriented software. Addison-Wesley, Reading, Mass.
- [Gosling et al., 2012] Gosling, J., Joy, B., Steele, G., Bracha, G. and Buckley, A. (2012). The Java Language Specification: Java SE 7 Edition. <http://docs.oracle.com/javase/specs/jls/se7/html/index.html>.
- [Hickey, 2008] Hickey, R. (2008). Clojure official website. <http://clojure.org>.
- [Kiczales et al., 1991] Kiczales, G., Rivièrs, J. d. and Bobrow, D. G. (1991). The Art of the Metaobject Protocol. The MIT Press, Cambridge; London.
- [Meyer, 1987] Meyer, B. (1987). Reusability: The Case for Object-Oriented Design. IEEE Software 4, 50–64.
- [Mezini et al., 2000] Mezini, M., Seiter, L. and Lieberherr, K. (2000). Component Integration with Pluggable Composite Adapters.

Appendix A

Boa results to HTML documents script

Listing A.1: Python script to produce HTML documents from Boa results. By Kardelen Hatun.

```
1 import os
2 from sys import stdin, stdout, stderr
3 import itertools
4
5 class Project:
6     def __init__(self):
7         self.url = ''
8         self.entries = list()
9
10 class Entry:
11     def __init__(self):
12         self.no = ''
13         self.comment = ''
14
15 def revisionURL(purl):
16
17     spliturl = purl.split('/')
18     pname = spliturl[-1]
19     url = 'http://' + spliturl[2] + '/' + 'viewvc' + '/' + pname
20         + '?view=revision&revision='
21     #print url
22     return url
23
24 def runQuery(searchstr, sameComment, projects):
25     filename = ''
26     for fnstr in searchstr:
27         filename += fnstr.strip(' ')
```

```

27
28 fn = "queryresult-2/" + filename + ".html"
29
30 if not os.path.exists(fn):
31     selProjects = dict()
32     for project in projects:
33         i = 0
34         temp = list()
35         for ent in project.entries:
36             if sameComment == 'True':
37                 flag = True
38                 for s in searchstr:
39                     if ent.comment.find(s) < 0:
40                         flag = False
41                 if flag:
42                     temp.append(ent)
43                     i+=1
44             else:
45                 for s in searchstr:
46                     if ent.comment.find(s) > 0:
47                         i+=1
48                     if ent not in temp:
49                         temp.append(ent)
50
51         if 0 < i:
52             project.entries = temp
53             selProjects[project] = i
54
55 sortedSelPro = sorted(selProjects.iteritems(), key=lambda
    (k,v): (v,k), reverse=False)
56
57 if len(sortedSelPro) > 0 and not os.path.exists(fn):
58
59     outfile = open(fn , 'w')
60     outfile.write('<html>')
61     outfile.write('<body bgcolor=\"white\">')
62     outfile.write('<table border=\"2\" width = \"100%\" >')
63
64     for project, j in sortedSelPro:
65         outfile.write('<tr>')
66         outfile.write( ' <td><a href=\"' + project.url + '\">'
            + project.url + '</a>' + '</td>')
67         outfile.write( ' <td>' + str(j) + '</td>')
68         outfile.write( ' <td>' )
69         outfile.write('<table>')

```



```

70     for ent in project.entries:
71         outfile.write('<tr>')
72         outfile.write(' <td>')
73         if project.url != '':
74             outfile.write(' <a href=\"' +
75                             revisionURL(project.url) + ent.no + '\"
76                             target=\"_blank\" >' + ent.no + '</a>')
77         else:
78             outfile.write(ent.no)
79             outfile.write('</td>')
80             outfile.write('</tr>')
81             outfile.write('<tr>')
82             outfile.write(' <td>' + ent.comment + '</td>')
83             outfile.write('</tr>')
84
85     outfile.write('</table>')
86     outfile.write(' </td>')
87     outfile.write('</tr>')
88
89     outfile.write('</table>')
90     outfile.write('</body>')
91     outfile.write('</html>')
92
93 file = open("bockisch-boa-output.txt")
94
95 projects = list()
96
97 for line in file:
98     ln = line.replace('\n', ' ')
99     for word in ln.split('['):
100         if word.endswith(']'):
101             w = word.strip('[]')
102             if len(projects) == 0:
103                 p = Project()
104                 p.url = w
105                 projects.append(p)
106             elif projects[-1].url != w:
107                 p = Project()
108                 p.url = w
109                 projects.append(p)
110         else:
111             e = Entry()
112             for wrd in word.split('='):
113                 if wrd.find(']') > 0:
114                     w2 = wrd.replace(']', ' ')

```

```

113         e.no = w2
114         #outfile.write(w2 + '\t')
115     else:
116         #outfile.write(wrd + '\t')
117         e.comment = wrd
118     projects[-1].entries.append(e)
119
120 query = {"adapter",
121         "integrat",
122         "plug",
123         "bind",
124         "depend",
125         "inject",
126         "wrap",
127         "glue",
128         "decouple",
129         "legacy",
130         "component",
131         "library"}
132 try:
133     os.mkdir("queryresult")
134 except:
135     pass
136
137 for pair in itertools.product(query, repeat=2):
138     runQuery(sorted(set(pair)), 'True', projects)

```

Appendix B

Validation source code

B.1 Hard-coded implementation

The application we have developed is a simple Java implementation of the unix `curl` or `wget` tool. Listing B.1 shows the `App` class and its `AppLogger`. We can see that it instantiates a logger itself on line 7, which is set to only display informational or more severe log messages. In line 8 it instantiates the `HTTP` component for use in the `App`. Listing B.2 shows the `HTTP` class and its `HTTPLogger`. We see that on line 19 it has a hard-coded dependency to its own logger, which is set to show debug messages on line 22. Running the application in this way yields the output as shown in listing B.3. As one can see, two loggers are used. The application logger uses capitals in the level of the message, whereas the `HTTP` logger prepends the level with “http-”. One also sees that debug messages are shown, even though the application logger is set to not display debugging messages. This is not an ideal situation.

Since the dependency on `HTTPLogger` by the `HTTP` class is hard-coded, and since we will not use any byte-code changing framework here, we will have to change its source to get to the situation where only one logger is used. One option would be to change the type of the `HTTP.logger` field to be an `AppLogger`. This would be very cumbersome though, since every call to the logger, which are already quite a few in our simple example as one can see, would need to be changed as well. A more sensible choice would be to introduce an `Adapter`.

Listing B.4 shows a suitable `Adapter`. Having this adapter, we can change the `HTTP` component to use the logger from the application with some changes. Listings B.5 and B.6 shows the updated part of both the `App` and `HTTP` classes. When we compile and run the application again, we see the desired logging output, as shown in listing B.7. Only the application logger is used, and no more debug messages are shown, as expected. Note that we used constructor dependency injection, to pass the correct logger to the `HTTP` component.

Listing B.1: Hardcoded - Bad situation - App component

```
1 package validation.simple;
2
3 import java.io.IOException;
4
5 public class App {
6
7     private AppLogger logger = new
        AppLogger(AppLogger.Level.INFO);
8     private HTTP http = new HTTP();
9
10    public App(final String[] args) {
11        boolean errorsOccured = false;
12
13        for(final String arg : args) {
14            logger.debug("Processing argument: " + arg);
15            try {
16                System.out.println(http.get(arg));
17            } catch (final IOException ioe) {
18                errorsOccured = true;
19            }
20        }
21
22        if (errorsOccured)
23            logger.error("Errors occurred, check log for details.");
24    }
25
26    public static void main(final String[] args) {
27        new App(args);
28    }
29 }
30
31 class AppLogger {
32
33     public static enum Level { DEBUG, INFO, WARN, ERROR }
34
35     private final Level level;
36
37     public AppLogger(Level level) {
38         this.level = level;
39     }
40
41     private void log(final Level lvl, final String msg) {
42         if (lvl.compareTo(level) >= 0)
```

```

43     System.err.println(String.format("[%s] %s", lvl, msg));
44 }
45
46 public void debug(final String msg) {
47     log(Level.DEBUG, msg);
48 }
49
50 public void info(final String msg) {
51     log(Level.INFO, msg);
52 }
53
54 public void warn(final String msg) {
55     log(Level.WARN, msg);
56 }
57
58 public void error(final String msg) {
59     log(Level.ERROR, msg);
60 }
61 }

```

Listing B.2: Hardcoded - Bad situation - HTTP component

```

1 package validation.simple;
2
3 import java.net.URL;
4 import java.net.URLConnection;
5
6 import java.util.Map;
7 import java.util.Map.Entry;
8 import java.util.List;
9
10 import java.io.IOException;
11 import java.io.InputStream;
12 import java.io.InputStreamReader;
13 import java.io.BufferedReader;
14 import java.io.StringWriter;
15 import java.io.PrintWriter;
16
17 public class HTTP {
18
19     protected HTTPLogger logger = new HTTPLogger();
20
21     public HTTP() {
22         logger.setLevel(HTTPLogger.LEVEL_DEBUG);
23     }

```

```

24
25 public String get(String urlStr) throws IOException {
26     logger.log(HTTPLogger.LEVEL_INFO, "Request to read from '"
27         + urlStr + "'");
28     InputStream in = null; // damn you Java, why not extend
29         the lexical scope of try into finally?
30     try {
31         logger.log(HTTPLogger.LEVEL_DEBUG, "Parsing URL '" +
32             urlStr + "'");
33         final URL url = new URL(urlStr);
34         logger.log(HTTPLogger.LEVEL_DEBUG, "Opening connection
35             to '" + url + "'");
36         final URLConnection urlc = url.openConnection();
37
38         logger.log(HTTPLogger.LEVEL_DEBUG, "Reading headers from
39             '" + url + "'");
40         final Map<String, List<String>> headers =
41             urlc.getHeaderFields();
42         for (Entry<String, List<String>> header :
43             headers.entrySet()) {
44             final String key = header.getKey();
45             final List<String> value = header.getValue();
46             if (key != null)
47                 logger.log(HTTPLogger.LEVEL_DEBUG, " header: '" +
48                     key + "' - '" + value + "'");
49         }
50
51         String result = null; // and why are blocks of code not
52             expressions? Argh..
53         if (headers.containsKey("Location")) {
54             final String newUrlStr =
55                 headers.get("Location").get(0);
56             logger.log(HTTPLogger.LEVEL_INFO, "Redirected from '"
57                 + urlStr + "' to '" + newUrlStr + "'");
58             result = get(newUrlStr);
59         } else {
60
61             logger.log(HTTPLogger.LEVEL_DEBUG, "Reading content
62                 from '" + url + "'");
63             in = urlc.getInputStream();
64             final InputStreamReader is = new InputStreamReader(in);
65             final StringBuilder sb = new StringBuilder();
66             final BufferedReader br = new BufferedReader(is);
67

```

```

57     String read = br.readLine();
58     while(read != null) {
59         sb.append(read);
60         read =br.readLine();
61     }
62
63     result = sb.toString();
64     logger.log(HTTPLogger.LEVEL_DEBUG, "Result from
        reading '" + url + "': " + result);
65     logger.log(HTTPLogger.LEVEL_INFO, "Done reading from
        '" + urlStr + "'.");
66 }
67
68     return result;
69 } catch (final IOException ioe) {
70     logger.log(HTTPLogger.LEVEL_ERROR,
71         HTTPLogger.exceptionToString(ioe));
72     throw ioe;
73 } finally {
74     if (in != null) {
75         logger.log(HTTPLogger.LEVEL_DEBUG, "Closing connection
76             to '" + urlStr + "'.");
77         in.close();
78     }
79 }
80
81 class HTTPLogger {
82
83     public static final int LEVEL_DEBUG = 0;
84     public static final int LEVEL_INFO = 1;
85     public static final int LEVEL_WARN = 2;
86     public static final int LEVEL_ERROR = 3;
87
88     protected static final String[] LEVEL_NAMES = {"debug",
89         "info", "warn", "error"};
90
91     protected int level = LEVEL_WARN;
92
93     public void setLevel(final int level) {
94         this.level = level;
95     }
96
97     public void log(final int level, final String message) {

```

```

97     if (this.level <= level)
98         System.err.println("[http-" + LEVEL_NAMES[level] + "]" +
99             + message);
100
101     }
102
103     public static String exceptionToString(final Exception ex) {
104         final StringWriter sw = new StringWriter();
105         final PrintWriter pw = new PrintWriter(sw);
106         ex.printStackTrace(pw);
107         return sw.toString();
108     }
109 }

```

Listing B.3: Output bad situation

```

1 $ java validation.simple.App http://ddg.gg > out.html
2 [http-info] Request to read from 'http://ddg.gg'.
3 [http-debug] Parsing URL 'http://ddg.gg'.
4 [http-debug] Opening connection to 'http://ddg.gg'.
5 [http-debug] Reading headers from 'http://ddg.gg'.
6 [http-debug] header: 'Date' - [Fri, 01 Feb 2013 17:32:14 GMT].
7 [http-debug] header: 'Content-Length' - [0].
8 [http-debug] header: 'Location' - [https://duckduckgo.com].
9 [http-debug] header: 'Server' - [Apache-Coyote/1.1].
10 [http-info] Redirected from 'http://ddg.gg' to
    'https://duckduckgo.com'.
11 [http-info] Request to read from 'https://duckduckgo.com'.
12 [http-debug] Parsing URL 'https://duckduckgo.com'.
13 [http-debug] Opening connection to 'https://duckduckgo.com'.
14 [http-debug] Reading headers from 'https://duckduckgo.com'.
15 [http-debug] header: 'Date' - [Fri, 01 Feb 2013 17:32:22 GMT].
16 [http-debug] header: 'Content-Length' - [7914].
17 [http-debug] header: 'Expires' - [Fri, 01 Feb 2013 17:32:23
    GMT].
18 [http-debug] header: 'Content-Type' - [text/html;
    charset=UTF-8].
19 [http-debug] header: 'Connection' - [keep-alive].
20 [http-debug] header: 'Accept-Ranges' - [bytes].
21 [http-debug] header: 'Server' - [nginx].
22 [http-debug] header: 'Cache-Control' - [max-age=1].
23 [http-debug] Reading content from 'https://duckduckgo.com'.
24 [http-debug] Result from reading 'https://duckduckgo.com':
    <!DOCTYPE -snip-
25 [http-info] Done reading from 'https://duckduckgo.com'.
26 [http-debug] Closing connection to 'https://duckduckgo.com'.

```



```

27
28 $ java validation.simple.App foo://ddg.gg > out.html
29 [http-info] Request to read from 'foo://ddg.gg'.
30 [http-debug] Parsing URL 'foo://ddg.gg'.
31 [http-error] java.net.MalformedURLException: unknown protocol:
    foo
32   at java.net.URL.<init>(URL.java:574)
33   at java.net.URL.<init>(URL.java:464)
34   at java.net.URL.<init>(URL.java:413)
35   at validation.simple.HTTP.get(HTTP.java:30)
36   at validation.simple.App.<init>(App.java:16)
37   at validation.simple.App.main(App.java:27)
38
39 [ERROR] Errors occurred, check log for details.

```

Listing B.4: Hardcoded - Integrated situation - Adapter

```

1 package validation.simple2;
2
3 public class App2HTTPLogger extends HTTPLogger {
4
5     private AppLogger adaptee;
6
7     public App2HTTPLogger(AppLogger adaptee) {
8         this.adaptee = adaptee;
9     }
10
11     public void setLevel(final int level) {
12         // making this a no-op.
13     }
14
15     public void log(final int level, final String message) {
16         switch (level) {
17             case 0: adaptee.debug(message); break;
18             case 1: adaptee.info(message); break;
19             case 2: adaptee.warn(message); break;
20             case 3: adaptee.error(message); break;
21         }
22     }
23 }

```

Listing B.5: Hardcoded - Integrated situation - App component

```

1 public class App {
2
3     private AppLogger logger = new

```

```

AppLogger(AppLogger.Level.INFO);
4 private HTTP http = new HTTP(logger);

```

Listing B.6: Hardcoded - Integrated situation - HTTP component

```

1 public class HTTP {
2
3     protected HTTPLogger logger = null;
4
5     public HTTP(AppLogger appLogger) {
6         logger = new App2HTTPLogger(appLogger);
7     }

```

Listing B.7: Output integrated situation

```

1 $ java validation.simple2.App http://ddg.gg > out.html
2 [INFO] Request to read from 'http://ddg.gg'.
3 [INFO] Redirected from 'http://ddg.gg' to
   'https://duckduckgo.com'.
4 [INFO] Request to read from 'https://duckduckgo.com'.
5 [INFO] Done reading from 'https://duckduckgo.com'.
6
7 $ java validation.simple2.App foo://ddg.gg > out.html
8 [INFO] Request to read from 'foo://ddg.gg'.
9 [ERROR] java.net.MalformedURLException: unknown protocol: foo
10   at java.net.URL.<init>(URL.java:574)
11   at java.net.URL.<init>(URL.java:464)
12   at java.net.URL.<init>(URL.java:413)
13   at validation.simple2.HTTP.get(HTTP.java:30)
14   at validation.simple2.App.<init>(App.java:16)
15   at validation.simple2.App.main(App.java:27)
16
17 [ERROR] Errors occurred, check log for details.

```

B.2 Google Guice implementation

Next, we took our base case from the former implementation, and changed it so both components use the popular Google Guice framework (version 3.0) for DI. The result for the App class can be seen in listing B.8. As one can see on line 3 and 4, the AppLogger is injected in the logger field. Unfortunately, Guice injects into this field *after* the object has been instantiated. This means that we had to replace the constructor with a method, in our case `run`. The instantiation of the App class has also changed, in order for Guice to perform the injections. This new instantiation can be seen on lines 24 to 27.

In Guice, the advertised way of specifying what to inject where, is by writing so-called *module* classes. One basically tells Guice which type to bind to another type or instance. The `@Inject` annotation tells Guice to inject something at that place, and Guice then injects that what is bound to the type of that place by looking in the registered binding modules. So, to inject an `AppLogger` into the `App.logger` field, we need a module as shown in listing B.9. In it, we see on line 10 that the `AppLogger` class is bound to an instance of it. The HTTP component needs a similar module, which is specified in listing B.11.

Now all we need to do to get to the preferred integrated situation is to change the binding in the `HTTPLoggerModule`. We want to bind the `HTTPLogger` class to an instance of an `AppLogger`. Since these are type incompatible, we need to use an adapter. We can reuse the adapter from the basic implementation, which gets us a module as shown in listing B.12.

Listing B.8: Guice - Bad situation - App component

```
1 public class App {
2
3     @Inject
4     private AppLogger logger;
5     private HTTP http = HTTP.getInstance();
6
7     public void run(final String[] args) {
8         boolean errorsOccured = false;
9
10        for(final String arg : args) {
11            logger.debug("Processing argument: " + arg);
12            try {
13                System.out.println(http.get(arg));
14            } catch (final IOException ioe) {
15                errorsOccured = true;
16            }
17        }
18
19        if (errorsOccured)
20            logger.error("Errors occurred, check log for details.");
21    }
22
23    public static void main(final String[] args) {
24        final Injector injector =
25            Guice.createInjector(new AppLoggerModule());
26        final App app = injector.getInstance(App.class);
27        app.run(args);
28    }
29 }
```

Listing B.9: Guice - Bad situation - App binding

```
1 package validation.guice;
2
3 import com.google.inject.AbstractModule;
4
5 public class AppLoggerModule extends AbstractModule {
6
7     public static AppLogger instance = new
        AppLogger(AppLogger.Level.DEBUG);
8
9     protected void configure() {
10         bind(AppLogger.class).toInstance(instance);
11     }
12 }
```

Listing B.10: Guice - Bad situation - HTTP component

```
1 public class HTTP {
2
3     @Inject
4     protected HTTPLogger logger;
5
6     public static HTTP getInstance() {
7         final Injector injector = Guice.createInjector(new
            HTTPLoggerModule());
8         final HTTP http = injector.getInstance(HTTP.class);
9         http.logger.setLevel(HTTPLogger.LEVEL_DEBUG);
10        return http;
11    }
```

Listing B.11: Guice - Bad situation - HTTP binding

```
1 package validation.guice;
2
3 import com.google.inject.AbstractModule;
4
5 public class HTTPLoggerModule extends AbstractModule {
6
7     protected HTTPLogger instance = new HTTPLogger();
8
9     protected void configure() {
10         bind(HTTPLogger.class).toInstance(instance);
11     }
12 }
```

Listing B.12: Guice - Integrated situation - HTTP binding

```

1 package validation.guice2;
2
3 import com.google.inject.AbstractModule;
4
5 public class HTTPLoggerModule extends AbstractModule {
6
7     protected void configure() {
8         bind(HTTPLogger.class).toInstance(new
9             App2HTTPLogger(AppLoggerModule.instance));
10    }
11 }

```

B.3 Gluer implementation

Now that two cases have been covered, we will describe our findings on applying our Gluer framework. For this, we took the base case again. This time it was our intention not to touch any existing source code, while still getting to the desired integration. What we want is to associate the `AppLogger` with the `HTTP.logger` field. This can be declared nicely in a `.gluer` file, as shown in listing B.13. Since the `AppLogger` class does not have a zero-argument constructor, we need to use a factory that returns a correctly instantiated logger. The factory itself can be seen in listing B.14. Again, the injection needs an adapter, for which we can reuse the `App2HTTPLogger` adapter. We only need to add the `@Adapter` annotation to it, in order for Gluer to recognise it.

Having this in place, one would expect to be done. No source was touched, and the injection will do the work. Unfortunately, the `HTTP.logger` field has an initialiser in the source, which Gluer cannot overwrite at the moment. Replacing this initialisation code once it is byte-code is hard. Therefore, we had to change the `HTTP` class a little, as one can see in listing B.16. If not for this change, the initialisation of the `logger` field would overwrite the injected object. With this change however, the integration works as desired.

Listing B.13: Gluer - Association

```

1 associate field validation.gluer.HTTP.logger with call
   validation.gluer.AppLoggerFactory.getLogger()

```

Listing B.14: Gluer - Factory

```

1 package validation.gluer;
2
3 public abstract class AppLoggerFactory {
4

```

```

5     private static AppLogger instance = new
        AppLogger(AppLogger.Level.WARN);
6
7     public static AppLogger getLogger() {
8         return instance;
9     }
10 }

```

Listing B.15: Gluer - Adapter

```

1 package validation.gluer;
2
3 import gluer.Adapter;
4
5 @Adapter
6 public class App2HTTPLogger extends HTTPLogger {

```

Listing B.16: Gluer - HTTP component

```

1 public class HTTP {
2
3     protected HTTPLogger logger;
4
5     public HTTP() {
6         if (logger == null) {
7             logger = new HTTPLogger();
8         }
9         logger.setLevel(HTTPLogger.LEVEL_DEBUG);
10    }

```

Listing B.17: Compilation and running with Gluer

```

1 $ javac -cp .:lib/gluer-0.1.0-SNAPSHOT-standalone.jar
    validation/gluer/*.java
2 $ java -cp .:lib/gluer-0.1.0-SNAPSHOT-standalone.jar
    gluer.core validation/gluer/gluer.config
3 No errors.
4
5 $ java -cp .
    -javaagent:lib/gluer-0.1.0-SNAPSHOT-standalone.jar=
    validation/gluer/gluer.config validation.gluer.App
    http://ddg.gg > out.html

```

B.4 Adapter resolution validation

The listing below is part of the source code of Gluer, and its role is to validate the resolution semantics. Note that tests that test the precedence declarations are missing. Precedence relations have been implemented at a later stadium, and time was lacking to add tests for them.

Listing B.18: Adapter resolution test suite

```
1 (deftest test-get-adapter-for
2   (with-redefs [r/leveled-supertypes-of #(supertypes %)]
3     (with-log-redefs []
4       (testing "Testing valid results."
5         (are [from to result] (= (:result (get-adapter-for
6           from to adapters)) result)
7           ; Exact match.
8           "B"      "A"      "BtoA"
9           ; Subtype of from.
10          "SubSubB" "A"      "SubBtoA"
11          ; Supertype of to.
12          "B"      "SuperSuperA" "BtoSuperA"
13          ; Subtype of from and supertype of to.
14          "SubB"    "SuperA"    "SubBtoA"
15          "SubSubB" "SuperSuperA" "SubBtoA"))
16       (testing "Testing errors."
17         (are [from to] (not (nil? (:error (get-adapter-for
18           from to adapters)))))
19         ; No match. No adapter to a type (or a subtype of
20         to).
21         "SubB"    "SubA"
22         ; No match. No adapter from a type (or a supertype
23         of from).
24         ; Might also be a warning, if adapters are found for
25         subtypes of from.
26         ; Or, this might be a reason for abstract Adapters.
27         "SuperB"  "SuperA"
28         ; Equal matches conflict.
29         "TwiceB"  "A"))
30       (testing "Testing warnings."
31         #_(are [from to] (not (nil? (:warning (get-adapter-for
32           from to adapters)))))
33         ; Possible runtime conflict error for subtype of
34         from.
35         "B"      "A"))))
```


Appendix C

Implementation details

C.1 Parsing

To meet the design goal of a dynamic parser as explained in section 2.4.2, and since the one candidate Clojure parser we found that met this goal was not to our liking, a new parser has been developed for use in Gluer.¹ The parser is a so-called PEG parser [Ford, 2004], and it has been developed as a separate library.² Since the parser is written in Clojure, some knowledge of the language is required to understand the description below of how it works.

The parsing rules are defined in a hash-map. Each entry defines a rule, where the key is a keyword denoting the rule's name and the value is the parsing expression. A parsing expression may be one of the following:

- A character, such as `\a`,
- A string, such as `"abc"`,
- A regular expression, such as `#"[a-c]+"`, or
- A vector, such as `[\a :b "c" / #"[d-z]+" [\+ / \-]]`.

The first three are considered terminals. A vector denotes a non-terminal sequence, and may be nested. A vector may contain any of the terminals, but also keywords to refer to other rules and the choice operator `/`. Any terminal in the vector is parsed, but do not end up in the resulting AST. As is standard in PEG parsers, the choices are prioritized, i.e. the first successfully parsed choice is used.

¹The other parser was called Parsley (<https://github.com/cgrand/parsley>), but the resulting AST was not easy to use nor idiomatic Clojure. The Gluer parser has been inspired by the "API" Parsley offers though.

²The library is called "Crustimoney" and can be found at <https://github.com/aroemers/crustimoney>.

C.1.1 Simple example

An example map of rules is the following:

```
1 (def calc
2   {:expr      [ :sum ]
3    :sum       [ :product :sum-op :sum / :product ]
4    :product   [ :value :product-op :product / :value ]
5    :value     [ :number / \( :expr \) ]
6    :sum-op    #"(\+|-) "
7    :product-op #"(\*|/) "
8    :number    #"[0-9]+"})
```

Above rules can parse simple arithmetic. Calling the `crustimoney.parse/parse` function with these rules and an expression would yield the following:

```
1 => (parse calc :expr "2+3-10*15")
2 {:succes
3  {:sum
4   [{:sum-op "+", :product {:value {:number "2"}}}
5    {:sum-op "-", :product {:value {:number "3"}}}]
6   {:product
7    [{:product-op "*", :value {:number "10"}}
8     {:value {:number "15"}}]}}}])
```

As one can see, the AST is a direct derivative of the parsing rules, except for the fact that recursive rules are nicely wrapped in a single vector, instead of being nested.

Note that PEG parsers have "greedy" parsing expressions by definition. This means that expressions cannot be left recursive. For example, a rule like `{:x [:x \a / \b]}` will never terminate. This is however a minor limitation, in return for clear parsing semantics, since every grammar can be rewritten to not being left recursive.

C.1.2 Non-terminals as terminals

Sometimes one wants terminals that cannot be defined by standard regular expressions, e.g. correctly nested parentheses. This can easily be defined using non-terminals, but this complicates the resulting AST. Therefore the library supports non-terminals that act like terminals. One achieves this by appending a `-` sign to the name of the rule. For example:

```
1 (def nested
```

```

2  { :root      [ :parens ]
3    :parens-   [ :non-paren :parens / :paren-open :parens
4      :paren-close :parens / ]
5    :non-paren  #"^[^\\(\\)]"
6    :paren-open  \(
7    :paren-close \) })

```

Above rule map parses any text, as long as the parentheses match correctly. But more importantly, the `:parens` part is regarded as a terminal, as can be seen when one parses an arbitrary expression:

```

1 => (parse nested :root "( (foo)bar(baz))woz")
2 { :succes { :parens "( (foo)bar(baz))woz" } }

```

C.1.3 Parse errors

In case one supplies an expression that cannot be parsed, the result is as follows:

```

1 => (parse calc :expr "2+3-10*") ; notice the missing part at
   the end.
2 { :error
3   { :errors #{"expected character '(' "
4     "expected a character sequence that matches
5       '[0-9]+' "},
6   :line 1, :column 8, :pos 7}}
7
8 => (parse nested :root "( (foo)bar(") ; notice the last paren.
9 { :error
10  { :errors #{"expected character ')' "
11    "expected character '(' "
12    "expected a character sequence that matches
13      '^[^\\(\\)]' "},
14  :line 1, :column 11, :pos 10}}

```

The `:errors` key contains a set of possible errors on the specified `:line` at the specified `:column`. The `:pos` key contains the overall character position of the errors in the text, starting at 0.

C.1.4 Whitespace

Whitespace needs to be defined explicitly in the grammar. The `crustimoney.parse/with-spaces` function is a small helper function for sequences that have mandatory whitespace between the items. For example:

```
1 (def hello
2   {:hello (with-spaces "hello" :name)
3    :name  #"[a-z]+"})
4
5 => (parse hello :hello "hello world")
6 {:succes {:name "world"}}
7
8 => (parse hello :hello "helloworld")
9 {:error
10  {:errors #{"expected a character sequence that matches
11             '\s+'"},
12   :line 1, :column 6, :pos 5}}
13
14 => (parse hello :hello "hello world ") ; notice the space at
15    the end.
16 {:error
17  {:errors #{"expected EOF"},
18   :line 1, :column 13, :pos 12}}
```

C.2 Adapter registry

The adapter registry holds all necessary information about the available adapters. First a list of classnames is retrieved, containing all the classes in the classpath annotated with `@Adapter`. Scanning the classpath, using a library called “Scannotation”, does not load the classes into the JVM, for it uses the “Javassist” library. The Javassist library reads class files and offers a model API. Based on this list of class names, the class models of them are read and the adapter registry can be created.

One function used when creating the registry, called `leveled-supertypes-of`, is of special interest here. This function takes a class name and returns a list of sets of class names. The first set in this list contains the direct supertypes of the specified class. The second set contains the direct supertypes of the classes in the first set. This continues like this, without repeating class names already in a previous set. Such a list makes it easy to determine the length of a path between two classes in a hierarchy.

The resulting registry is a map, where the key is the adapter class name. The value is again a map with two keys:

:adapts-from The value under this key contains a set with the class names that the adapter takes as adaptees.

:adapts-to The value under this key is the result of calling `leveled-supertypes-of` with the adapter class as its parameter.

In Clojure, a literal representation would be as follows:

```
{ "an.Adapter" { :adapts-from #{ "some.Adaptee" "other.Adaptee" }
                 :adapts-to   (#{ "a.SuperClass" "an.Interface" }
                               #{ "a.SuperInterface"
                                   "java.lang.Object" }) }
  "another.Adapter" ... }
```

C.3 Precedence relations

Another data structure in the implementation is the map holding the precedence relations. It is easily constructed by applying some transformations on parts of the AST given by the parser. The key in the map is the class name of an adapter that is preceded by other adapters. The value under such a key is a set with the names of the adapter that precede the adapter in the key. As explained in section 2.1.3, one can view these relations as a directed graph. Each key is a vertex, and the value contains the directed edges to other vertices.³ A literal representation would be as follows:

```
{ "adapter.C" #{ "adapter.A" "adapter.B" }
  "adapter.B" #{ "adapter.A" } }
```

The algorithm for finding a cycle is relatively simple. One picks a random key from the map, and recursively visit all the keys in the value of that key, while accumulating the keys that are visited. If a key would be visited for a second time, a cycle is detected. If a cycle has been detected or all “edges” have been visited, one repeats this process on a map with the keys that have been visited removed, for the graph may very well have multiple components, until the map is empty.

³Note that keys of these other vertices need not be present in the data structure, if it is not preceded by another adapter.

Appendix D

List of acronyms

AOP Aspect-Oriented Programming

API Application Programming Interface

AST Abstract Syntax Tree

CLOS Common Lisp Object System

DI Dependency Injection

DSL Domain-Specific Language

FP Functional Programming

FQN Fully Qualified Name

HTTP HyperText Transfer Protocol

IoC Inversion of Control

JAR Java Archive

JVM Java Virtual Machine

LP Logic Programming

OOP Object-Oriented Programming

PEG Parsing Expression Grammar

URL Unified Resource Locator

VCS Version Control System