NON-INTRUSIVE INSTANCE LEVEL SOFTWARE COMPOSITION

Kardelen Hatun

NON-INTRUSIVE INSTANCE LEVEL SOFTWARE COMPOSITION

Kardelen Hatun

Ph.D. Dissertation Committee

Chairman and Secretary

Prof. Dr. Ir. A.J. Mouthaan	University of Twente, The Netherlands

Promotor

Prof. Dr. Ir. Mehmet Akşit	University of Twente, The Netherlands

Assistant Promotor

Dr.-Ing. Christoph Bockisch	University of Twente, The Netherlands

Members

Prof. Dr. Ir. Wouter Joosen	Katholieke Universiteit Leuven, Belgium
Prof. Dr.-Ing. Mira Mezini	Technische Universität Darmstadt, Germany
Dr. Luís Ferreira Pires	University of Twente, The Netherlands
Prof. Dr. Roel J. Wieringa	University of Twente, The Netherlands

CTIT Ph.D. Thesis Series No. is 14-296
ISSN 1381-3617
Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE
Enschede, The Netherlands.

This work has been carried out as part of the OCTOPUS project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute program.

ISBN 978-90-365-3619-6
ISSN 1381-3617 (CTIT Ph.D. Thesis Series No. is 14-296)
DOI 10.3990/1.9789036536196

Cover Design: Kardelen Hatun
LaTeX template *classicthesis* by André Miede
Printed by Ipskamp Drukkers B.V. Enschede, The Netherlands

NON-INTRUSIVE INSTANCE LEVEL SOFTWARE COMPOSITION

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof.dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Wednesday, 12 February 2014 at 14.45

by

KARDELEN HATUN
born on 6 September 1984
in Ankara, Turkey

This thesis has been approved by:
Prof. Dr. Ir. Mehmet Akşit (promotor)
Dr.-Ing. Christoph Bockisch (assistant promotor)

This thesis is dedicated to...

my mother *Nazife*,
Because I feel whole
when I remember I'm *her* daughter...

my father *Şükrü*,
Because *he* taught me to be my own person...

my sister *Zeynep*,
Because *she* is all that is good in the world.

ACKNOWLEDGEMENTS

Doing a PhD is indeed a transforming experience. In the process I had to deconstruct and reconstruct everything I knew about myself, my abilities and other people. Now it's coming to an end and I would like to thank the people who were with me during this journey.

I would like to thank Mehmet Akşit, for giving me the opportunity to pursue my studies in TRESE. I am very grateful that you've pushed me further with your demand for quality research.

This thesis would have been a distant dream without Christoph Bockisch. Christoph, your perspective on research (and life in general) motivated me immensely. You are one of those rare people who can find order in chaos (e.g. your office), which was exactly what I needed. Please know that, you've helped me become a better version of myself. I will be forever grateful for your mentorship.

I feel honoured that Mira Mezini, Wouter Joosen, Luís Pires and Roel Wieringa agreed to be on my committee and improved this thesis with their comments.

I would like to express my sincere gratitude to Jeanette, the supersecretary, for her kind heart, friendly face and for making life so much easier for everyone around her.

Of course these five years would be so boring without Hayrettin. Thanks to our enormous database of inside jokes, we can make each other laugh with a single word, which is of course, delivered with perfect comedic timing. I really feel lucky to have a person whose sense of humour is equally twisted, as a friend.

My life in Enschede is filled with so much joy thanks to my dear friends; Alim, Ansfrida, Arda, Duygu, Eda, Emre, Engin Torun, Gülistan, Hasan, Muharrem, Özlem Sukaş, Ramazan, Recep, Sertan, Sinem, Umut and Yakup. I'm grateful to Didem and Semih for always being

there for me and for inviting me to the one of the most amazing experiences I've ever had in my life; the birth of Tunç. I've also had the thrill of experiencing the birth of Asya Roza, I thank Özlem and Engin for including me in this lovely memory and for providing the most "muhabbet ortam" ever. My warm feelings to Andreea, Christian and Lisa, my favourite companions for the Christmas market! I'd like to thank my colleagues and friends in TRESE; Selim, Hasan, Gürcan, Wilke, Haihan, Somayeh and Steven. I'd like to extend special thanks to Arjan de Roo, who have been my road buddy for many years. He drove diligently, even though I was dozing off in the passenger seat most of the time. I'd like to thank my one and only Master student, Arnout Roemers for the interesting conversations and his contribution to my research. Many thanks to RAM Board Game Night group for fun times and for introducing me to the fascinating world of board games. I'd like to thank my colleagues at Nedap for their support during my thesis writing process.

My old friends Selda, Aylin, Sakal, Emrah, Emre, Çağrı, Muzo, Büşra, Uğur, Alper, Tirben and Elif; even though we have been far away all these years; I always know that when we see each other, we will pick up where we left off. You have a place in my heart forever.

I feel extremely lucky that I've become a part of the Oğuz family, my mother-in-law Gönül, father in-law Bircan and sister-in-law Nazlı. You have made me feel as a part of your family from the first moment I've met you. Our wedding was one of the happiest days in my life and that is all thanks to you.

My fabulous uncle Cengiz, thank you for cooking fish for us, loving us and taking care of us. You know I love you very much.

And my partner in crime, Oğuzcan... Life is so much more interesting when I'm by your side. You're my favourite person to do everything with!

Kardelen
Janurary 2014
Enschede

ABSTRACT

A software system is comprised of parts, which interact through shared interfaces. Certain qualities of integration, such as loose-coupling, requiring minimal changes to the software and fine-grained localisation of dependencies, have impact on the overall software quality. Current general-purpose languages do not have features that target these integration qualities at the instance level, hence they lack expressive power to define integration-specific code. As a result integration requires invasive code alterations, tightly coupled components that hinder maintainability and reuse.

In this thesis, we focus on developing language extensions and frameworks, which offer a declarative way of defining the elements involved in the instance level software composition. Our motivation is that non-intrusive means of integration at the granularity of the instance level has an impact on the maintainability and the extensibility of the software. We focused on declarativeness since we want to improve how integration concerns are expressed in the implementation.

We particularly focus on two challenges specific to the instance-level integration step; the two contributions proposed as a solution to each of these challenges both present declarative approaches for implementing specific concerns. These concerns are; 1. selecting objects based on how they are used in a system and, 2. non-intrusive implementation and injection of adapters.

The first challenge is the difficulty of selecting objects based on other criteria than the type system. This is important during integration since, independent of their type, objects can become relevant to a component when they participate in specific events. Such events mark the phases in the life-cycle of objects. The phase in which an object currently is, affects how it is handled in an application; however phase shifts are often

implicit. Selecting objects according to such phase shifts results in scattered and tangled code. To handle these problems, we introduce a novel aspect-oriented concept, called *instance pointcuts*, for maintaining sets that contain objects with a specified usage history. Specifics are provided in terms of pointcut-like declarations selecting events in the life-cycle of objects. Instance pointcuts can be reused, by refining their selection criteria, e.g., by restricting the scope of an existing instance pointcut; and they can be composed, e.g., by set operations. These features make instance pointcuts easy to evolve according to new requirements. The instance pointcuts approach adds a new dimension to modularity by providing a fine-grained mechanism and a declarative syntax to create and maintain phase-specific object sets.

The second challenge we have tackled is establishing common interfaces between instances while maintaining loose coupling. To this end we have created an adaptation framework, called `zamk`, which unites dependency injection with *under-the-hood* adaptation logic. Due to limitations we have identified in the traditional adapter pattern, such as an increased number of dependencies and implementation challenges due to dependence on type inheritance, we have created the concept of *converters*, which are annotated classes that adhere to a specific structure. Converter classes do not have to inherit from other classes to implement the adaptation logic. They are defined by the user and managed by the `zamk` runtime; consequently the only dependency that needs to be introduced during integration is calls to the `zamk` API. `zamk` comes with its own dependency injection mechanism that is used with a designated domain-specific language called *Gluer*. The dependency injection logic is intertwined with the adaptation logic which queries a registry of converters to perform automated adaptation between two types. We automate the adaptation process by exploiting the type hierarchies and provide checks and context-relevant messages for correct integration. As a result the `zamk` framework provides a non-intrusive approach for adapting and binding software, which supports code reuse, software maintainability and evolution.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ACRONYMS

OOP Object-Oriented Programming

OO Object-Oriented

AOP Aspect-Oriented Programming

AO Aspect-Oriented

CF Composition Filters

DI Dependency Injection

Part I

INTRODUCTION

In this part we define the scope of this thesis and describe problem statements in detail. This thesis focuses on the problems that arise due to lack of modularity in the specification of related objects and missing support for object-level interactions. We tackle these problems with two novel approaches. The first problem is related to the type-based categorisation of objects in current programming languages. The second problem we present is the insufficient support for non-intrusively implementing and injecting object adapters. This part also discusses a small case study, which was our source of inspiration to start investigating the problems addressed in this thesis.

1

OVERVIEW

Complex software consists of many parts which are connected to each other to create a functional unit. Often, we would like to reuse software parts to create reliable software in a fast and a cost effective manner [Kru92]. Software reuse requires establishing connections between these parts [Szy02]; which may not share a common interface to communicate due to, for example, evolution of software with unplanned functionality, development of software parts independently of each other and/or introduction of design mistakes.

If software must be extended with unanticipated functionality, reuse is even more difficult. In such cases integrating this new functionality may require invasive code alterations. First, the new functionality may be in need of data which has not been made accessible by the software. In order to allow access, the software must be changed to expose this data. However, it may not be desirable (e.g, for architectural reasons) or even possible (e.g. source code is not available) to change the implementation. Second, since the parts rely on interfaces for interacting with each other, the incompatibility of interfaces causes an integration problem as well. To remedy interface incompatibility, developers can introduce new code to the software which must be maintained in case of subsequent software evolutions. Some approaches rely on immutable interfaces to protect integration code from software evolution. However this creates bottlenecks in terms of exposed features; when the behaviour of a component is extended, it almost always requires interface extensions [Spa00].

Since object-oriented languages [RBL+90] are very popular, from now on by means of software parts, we will refer to objects as in Object-Oriented Programming (OOP), unless stated otherwise. In OOP, objects are instances of their classes. The information which indicates to which class an object belongs is called the type information. Classes have an interface, which is a contract between a class and the outside world. Objects can only expose their state through their interfaces. In order to obtain information about an object, one has to use its interface. For two objects to communicate, they should have compatible interfaces. One of the ways to provide compatibility between interfaces is to introduce an intermediate type, which is called an adapter. Much like a real-world power adapter, the adapter types convert the interface of one type to the interface of the other.

From the above discussion, we identify two specific integration challenges for Object-Oriented (OO) languages:

(a) Proper integration of objects within a program may require objects to access each other's internal data values. Current languages only provide features to allow objects query on other objects based on their types, however it is also important to select objects based on how they are used in an application.

(b) When integrating new functionality into existing software, the instance-level integration code tends to be ad-hoc and fragile. Such code often consists of a collection of adapters and references to these adapters from different software parts. This type of integration code is not robust against software evolution.

In this thesis we improve on these challenges; our solutions in particular focus on non-intrusive integration mechanisms. For the first challenge we introduce "instance pointcuts", which is an aspect-oriented language extension to select objects based on their usage history. Instance pointcuts provide means to categorise objects according to how they are used in an application.

For the second challenge we have developed an adaptation-aware dependency injection framework, called *zamk*[1], which localises the integration code and introduces a lightweight way of implementing adaptations between incompatible interfaces as structures which we call *converters*.

1.1 STATE OF THE ART

The contributions presented in this thesis focus on non-intrusiveness. Two technologies that aim at mitigating the two identified challenges are Aspect-Oriented Programming (AOP) and Dependency Injection (DI); we discuss these in the following.

1.1.1 *Aspect-Oriented Programming (AOP)*

Separation of concerns [Dij82] is possibly the most important principle of software engineering; it is the basis for many software design practices such as modularity, reusability and maintenance. Separation of concerns is a design principle which modularises software into separate parts, each of which address a particular problem [Par72]. Ideally a concern should be implemented in a single unit, however in complex software there are various concerns which are hard to modularise with the language concepts provided by object-oriented programming. The concerns which cannot be contained in a unit crosscut other concerns; a crosscutting concern is an extra concern which disrupts the implementation of the main concern of a program. A crosscutting concern can be tangled and scattered. Tangling occurs when the crosscutting concern is implemented together with another concern in a compilation unit. Scattering occurs when a single concern is implemented in parts among many compilation units.

AOP is a paradigm which aims to improve separation of concerns by modularising crosscutting concerns in separate modules called aspects.

1 (Turkish) A type of plant gum which has sticky or adhesive quality.

Aspect-Oriented (AO) languages provide mechanisms for intercepting calls that are made in an application. Using the context information in an intercepted call, aspects define additional operations to be executed in the vicinity of the call; these operations are encapsulated in the body of an advice.

In the literature there are numerous studies which bring AOP and integration together. The reason is that AOP provides another dimension of modularisation, which is crucial for the non-invasive composition of software [TOHSJ99].

In terms of integration, an interesting property of CaesarJ [AGMO06, AGMO13] is having *wrappers* as a first-class language feature. Wrappers are a built-in adaptation/composition mechanism, whose state is contained in the *wrapee* object. Since CaesarJ uses AspectJ pointcut mechanisms for selecting wrapees, it does not provide a modular mechanism for maintaining wrapee collections. JAsCo [SVJ03, SV13] is another AO-language that is tailored for component-based development. JAsCo introduces *aspect beans* and *connectors*. Connectors can deploy *hooks* (similar to advices) depending on the context. By separating the aspect implementation from aspect binding, JAsCo provides improved expressiveness for integrating components. JAsCo does not provide language abstractions for selecting objects based on their use either. Aspectual Collaborations [LLO03], which builds on pluggable adapters [MSUL99] is another study which exploits aspect-orientation for integration of software parts. Using Aspectual Collaborations, one is able to define connections between objects by declaring dependencies and creating adapters.

1.1.2 *Dependency Injection (DI)*

Dependency Injection (DI) [Fow04], also referred to as Inversion of Control, is a way of establishing loose-coupling between software parts. A type is said to have a dependency to another type, if that type uses that other type. DI eliminates dependencies to *specific* types; the underlying implementation of a field can be initialised with different implementa-

tions of the same abstraction. The module which is responsible for injecting dependencies is called an injector. Injectors localise the configuration code for the software therefore they make it possible to change the configuration of the software without changing the dependent classes. DI is also beneficial for testing [MFC01]. Since we can inject dependencies to components, we can also inject mock implementations with various configurations to efficiently test software.

There are multiple ways to perform dependency injection; these are constructor injection, setter injection and interface injection. In constructor injection, the dependent field is populated in the constructor, by passing the specific object as a constructor parameter. In setter injection, the field is populated by calling the setter of that field with the injected value. In interface injection, the injection points are determined by finding the implementors of the declared injection interfaces.

DI is widely used in software development today. Google Guice [Goo13] is a light weight dependency injection framework, which uses annotations to mark the injection points. Developers configure the interface and the corresponding class binding using the Guice API; these bindings represent the configuration of the software. The problem with using framework-specific annotations is the coupling of the code with a specific framework. Spring Framework [JHAT09, Fra13] is a platform for developing enterprise applications, which also relies on DI for composition and configuration. The injection points can be declared with annotations or can be defined in XML as a portable format. A disadvantage of using the XML format is the lack of compile-time checking. PicoContainer [pic13] is one of the earliest DI frameworks and it does not require annotations or external files. Instead the developer must register the injection points and injectibles to the container; the PicoContainer handles the injection automatically.

1.2 APPROACH

Modularisation of objects based on their participation in certain interesting events

In Chapter 3 we look at component integration at the instance level. Software parts communicate data using objects; however an object may not become relevant until it participates in a certain event. In the control flow of an application such events can be implicit, e.g, the object may be passed as an argument. Also events may not change the object hence the occurrence of the event is not reflected in the object state. So the static information about objects, such as its type is not always sufficient to select relevant ones. However, the event information may be scattered around the application, requiring invasive code alterations to identify relevant objects; so the object selection concern becomes crosscutting.

In Chapter 3 we describe a new AO-language construct for modularising object selection. This construct, called instance pointcuts, reifies a set of objects of the same type hierarchy. These objects are selected using pointcut like notation, according to the events they participate in. With instance pointcuts we can create a module as a set of objects, which are in a relevant period in their life-cycle, where the beginning and the end of a period is marked by events. Instance pointcuts are composable by set operations and they can be reused to create other instance pointcuts.

In Chapter 3 we discuss instance pointcuts in detail starting with a motivating example and then moving onto instance pointcuts' syntax, semantics and possible applications. We also explain our modular code generation approach in detail.

A dependency injection framework with under-the-hood adaptation logic for binding components

In Chapter 5 we explore interface incompatibility and instance level dependencies in software composition. Objects require a shared interface

to interoperate, which is usually not present when integrating them later in the life-cycle of a software. This requires adapting interfaces of objects so that they can communicate. Another aspect of object communication is dependencies; explicit dependencies to concrete types result in tightly coupled software. When these two issues come together, object integration requires creating dependency to an adapter object.

In Chapter 5 we present our contribution, the *zamk* framework, which improves the integration process of externally developed software. For this framework we have developed an external DI language called *Gluer*. This language is used in conjunction with *converters*, which are reinterpreted adapters used for converting objects of one type to another type. *Gluer* is a converter-aware language meaning it can convert injected objects into other types before the injection, given the target injection field is of an incompatible type. Using *zamk* framework developers are able to separate the concerns of the software integration into adaptation and gluing.

ORGANISATION OF THE THESIS

In Chapter 2 we set the focus of this thesis and explain problems of object-level modularity and interaction by means of related work and personal experience. In this chapter we present a conceptual printing application, which was developed as a part of a project. We present a scenario where a scheduler software of a printing machine should be replaced and evolved. While explaining the scenario we point out the problems which were discussed at the beginning of the chapter.

In Chapter 3 and Chapter 4 we present the instance pointcuts approach which localises the object selection concern in a modular way.

In Chapter 5 and Chapter 6 we present our adapter-aware dependency injection framework *zamk*, which performs automatic adaptation before injecting values to target injection points.

In the final chapter, Chapter 7, we summarise our contributions and elaborate on future work.

2

MOTIVATION AND CONTEXT

The key element of object-oriented programming are objects; objects consist of a state which is stored in fields and a behavior which is exposed through methods. As software systems became more complex, features offered by Object-Oriented Programming (OOP) proved to be insufficient and led to the design of new programming paradigms such as Aspect-Oriented Programming (AOP) (briefly introduced in Chapter 1). This thesis focuses on the problems regarding object-level modularity and interaction. These two qualities play a central role in software evolution, where expressive power over accessing and using the data in the legacy software is of crucial importance. We particularly specialise on *unplanned* software evolution, where the implementation of *unanticipated concerns* is required. Therefore we focus on developing non-intrusive approaches for supporting the implementation of such concerns.

2.1 OBJECT INTERACTION

When new functionality is required in a software system, it is often cheaper to re-use existing components which provide this functionality [BCK03]. Component-based approaches for integrating components uniformly and efficiently, such as Component Models [LW07] and Component Frameworks [Vin97, RB10], offer an architectural solution. These techniques are therefore suitable for enterprise software and are usually adopted during design time.

Integrating new features to the software after its release has different requirements. Such software has legacy code and it is usually risky to make alterations to this code due to the possible introduction of bugs [Leh96]. In this context, a potentially problematic software evolution scenario is the integration of unanticipated functionality. This is because the legacy software may not provide the correct interfaces or contain the relevant dependencies to host this functionality. In addition, using third-party software comes with its own problems; third-party software evolution cannot be controlled and its integration requires further maintenance efforts [BA99, YBB99].

Interface compatibility can be established by using the adapter pattern [GHJV95]. Adapters can be used to plug in software parts into a software system, hence facilitating software reuse. However, the traditional adapter pattern has some drawbacks. First, the adapter type is an additional type that is introduced to the system and to the type hierarchy of the existing classes. Therefore using adapters may introduce unintended complexity to the software. Second, instantiating the adapter objects and adding references to these objects introduce new dependencies in the software. These observations led us to the conclusion that we need an approach which provides interface compatibility that does not complicate the type hierarchies and allows creating dependencies in a non-intrusive manner.

2.2 OBJECT-LEVEL MODULARITY

In OOP objects are categorised by their types. Type-based categorisation provides limited perspective on object roles; this was observed in early work on OOP [Kri96]. A statement that is added to a class definition affects all the instance of that class. AOP languages also operate with the type criteria; statements in an aspect will affect all the instances of the advised type. Rajan and Sullivan also point out a similar problem in [RS03b].

Categorising objects based on criteria other than their type and providing object-level advice has been the subject of many studies. In AspectS [Hir03], Hirschfeld proposes an AO-language which provides a more granular way of selecting instances of particular types, for example, by checking if a type implements a certain method. Although this approach provides more granularity, it cannot select objects based on events. In Eos [RS03a] a new AO with a rich join-point model to allow object-level advising is proposed. In CaesarJ [AGMO06], *wrappers* are applied to particular objects, which are selected by AspectJ pointcuts. In [SMU+06], Sakurai et al. also recognised the importance of associating objects for object-level advice and introduced association aspects. In this work an aspect is declared `perobject` and with `associate` method aspect are instantiated to objects. This is limiting since the developer is not flexible in associating objects with events other than a call to the `associate` method.

Composition Filters (CF) [AWB+94, BA01a] provide a different approach by presenting a new object model (Figure 2.1, taken from [BNGA04]). In composition filters object model, there is an inner and an outer layer. The inner layer is a plain object which is referred to as *kernel object*, and the outer layer that wraps that object is called the interface part. In CF, filters are superimposed on objects which filter the received and sent messages. This kind of control over objects inherently provides mechanisms for object-level advising. Since composition filters are superimposed per object, they do not provide a way to select a collection of objects.

So far we have established that object-level advising requires means to select relevant objects; in current languages this is supported through the type system (with the exception of association aspects [SMU+06], which use the calling of the `associate` as the method for selection). In this thesis we are interested in non-type based modularisation of a set of objects because of its value in software evolution scenarios. The type information alone does not give enough information about an object's role or use in the application.

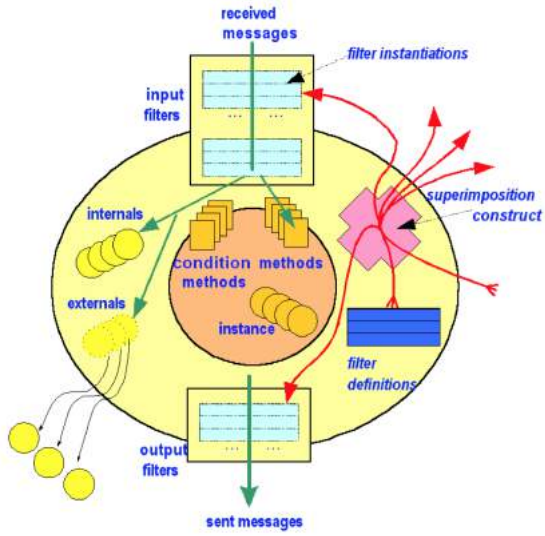


Figure 2.1: Composition Filters object model

All of the work mentioned so far either offer finer granularity in selecting objects thorough richer pointcut declarations or mention the importance of object-level advising, but do not propose an object selection mechanism which is reusable. These studies focus on join-point richness, which is important yet is not enough. This observation led us to the following problem statement: selecting objects according to the roles they play in an application is not declaratively supported by current languages, and this selection concern is not supported as a reusable language construct.

2.3 ILLUSTRATIVE CASE STUDY

In a previous research project [1] we were presented with the problem of evolving the scheduling[2] component of a heavy duty printing machine. In a printer the timing of the physical printing tasks has the utmost importance; which led the developers to implement a conservative scheduler. With further advances in their hardware, this scheduler proved to be a bottleneck for performance. They indicated that the new scheduling software must be able to use multiple scheduling strategies and must be able to handle print jobs in a more efficient way according to the resources of the printer.

The illustrative example we present in this section is inspired by this industrial case study; we use this example to elaborate on the software composition problems we have presented so far. The details of the actual case study can be seen in [HdRB+13].

2.3.1 *Problem Setting*

Assume that a *scheduling library* is adopted for implementing the new scheduler. This library contains abstractions for scheduling concepts like resources, demands, tasks etc. Another library, which contains various scheduling algorithms, is also selected to be used with the scheduling library.

In this problem setting we are confronted with the following challenges:

- Adapting the legacy printer software to work with the new scheduling library.
- Establishing dependencies between existing software and the new scheduler.

1 See http://www.esi.nl/octopus/

2 Scheduling is the process of *allocating resources to jobs over a period of time while optimising one or more objectives* [Bru01].

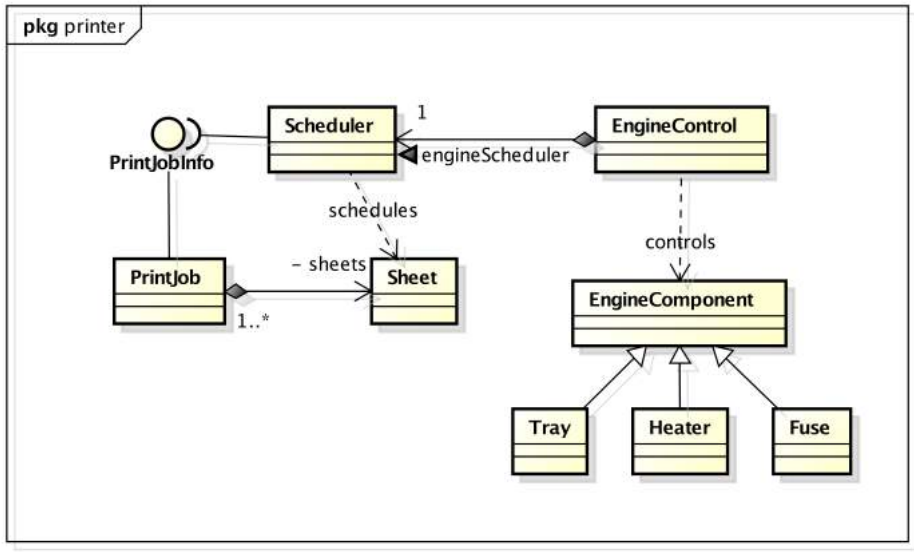


Figure 2.2: The static structure of the legacy printer

- Extracting task and resource data to be used by the scheduling algorithms library.

Legacy Software

The legacy class structure of the printing software can be seen in Figure 2.2. Each print job that is sent is a collection of sheets of paper, therefore the scheduler component schedules sheets. The scheduler asks the engine components to find an available time slot for scheduling the next sheet (Figure 2.3). Each engine component contains its own schedule information; by asking the first engine component, tray, the scheduler triggers a sequence of messages. Each engine component forwards the availability request to the next one. This schedule is later used by the engine control to execute the actions in a timely fashion to print a sheet of paper. In the printer, a sheet of paper has a specific path it should follow

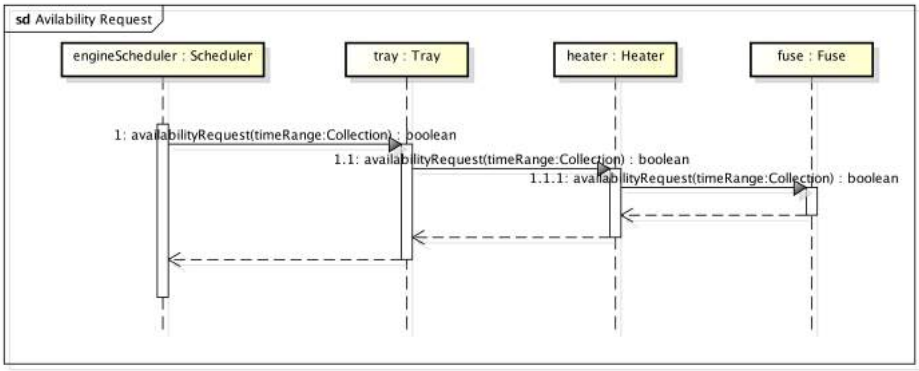


Figure 2.3: Sequence Diagram of scheduler's availability request

to be printed on; it is first separated from the tray, then it is heated to get to the optimal temperature for printing and finally in the fuse the sheet is sprayed with ink.

Scheduling Library

The scheduling library has its own static structure for implementing a scheduler (Figure 2.4). In this structure, the scheduler has references to available resources in the system. Each task has a specific resource demand; this demand is used to schedule these tasks. Once a schedule is calculated, it is communicated through the `SchedulingInfo` interface. In this scheduling library it is possible to attach external scheduling algorithms, given that they implement the `SchedulingAlgorithm` interface the scheduler component expects.

Notice that the scheduler structure this library imposes is different than the legacy structure shown in Figure 2.2. In the library structure all the information about scheduling is centralised in the scheduler component. In the legacy structure the engine components have the scheduling information and they share this information with the scheduler through method calls.

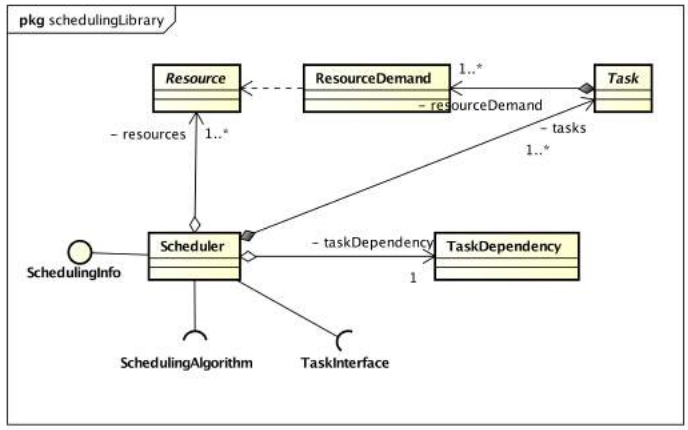


Figure 2.4: The static structure of the scheduling library

Scheduling Algorithms Library

The scheduling algorithm library expects a "scheduling problem" as input. Then the chosen scheduling algorithm, also taking the current schedule into account, outputs a solution (i.e a schedule) to the given scheduling problem. The scheduling problem is defined in terms of the $\alpha \mid \beta \mid \gamma$ notation defined in [GLLRK77]; these are the machine environment, task characteristics and the objective function. The static structure of the required interfaces to use this library can be seen in Figure 2.5.

In order to use this scheduling algorithms library, we must make the information needed explicit to model the scheduling problem at hand.

2.3.2 *Scenario 1: Integrating the new scheduler*

In this scenario we discuss the steps involved in integrating the new scheduler component to the legacy system. Note that this scenario does not include the integration of the scheduling algorithm. In Figure 2.6, we show a possible static structure after the integration. The existing

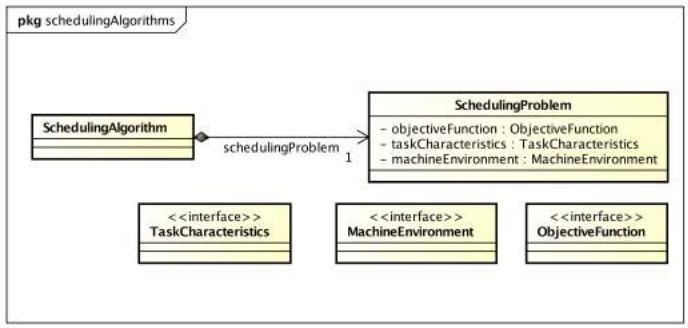


Figure 2.5: The static structure of the scheduling algorithms library

structure, `EngineComponent` and `Sheet` are adapted to `Resource` and `Task` respectively in order to be used by the new scheduler. The `PrintJob` legacy class now implements the interface `TaskInterface` of the scheduling library, which is implemented to convert print jobs to multiple tasks. In the new structure, each sheet of paper is separated into three tasks that correspond to each of the resources; tray separation task, heating task and fusing task. Each of these tasks demands a specific resource, which can be deduced from their naming. For brevity we have left these structures out of the figure. The class which is responsible for executing the tasks for each engine component now uses the interface `SchedulingInfo` to obtain the schedule of the tasks. In this implementation the individual engine components are not responsible for communicating scheduling information anymore; all of that information is centralised in the new scheduler component.

Even in this simplistic view, we had to introduce two new types (adapters), changed the type hierarchy of the legacy system and altered the dependencies in multiple classes. Also, this view assumes that there is a one-to-one mapping between the adapted types, however this may not always be the case. The adaptation may require other information than the one provided by the legacy objects. Another potential problem is

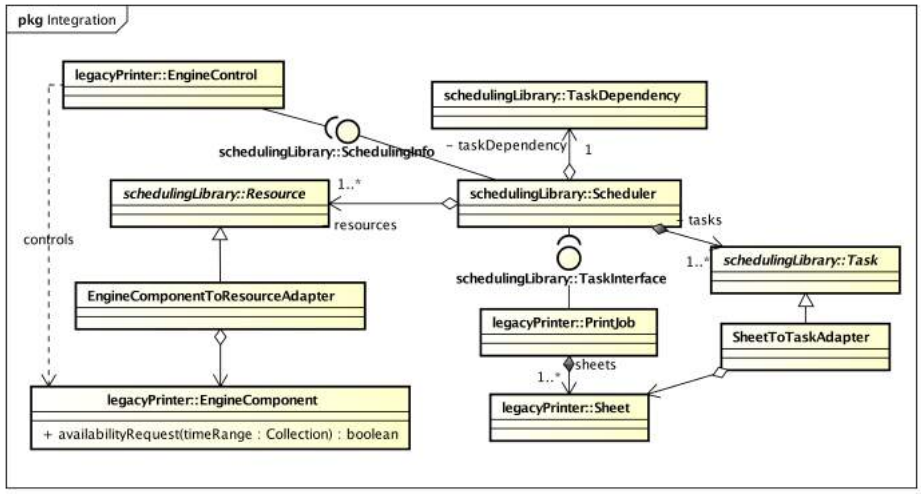


Figure 2.6: Scenario 1: Integrating the new scheduler

the injection of the adapter instances. The `EngineControl` class, which is the manager for the scheduler and engine component objects, will be responsible for creating the adapters and passing them on to the relevant objects. These challenges are summarized in our problem statement on object interaction (Section 2.1).

2.3.3 *Scenario 2: Integrating the Scheduling Algorithm*

In Figure 2.7 we show the integration of the scheduling algorithm with the new scheduler. As we have stated at the beginning of this case study, we would like to support multiple scheduling algorithms according to the state of the printer. For example, if the printer is being operated in an especially cold room, the heating operation may take longer. In this case the printer may decide to prioritise print jobs that use lighter paper, which is easier to heat. Another example is when a finisher component is attached to the printer, in order to bind or staple the sheets together;

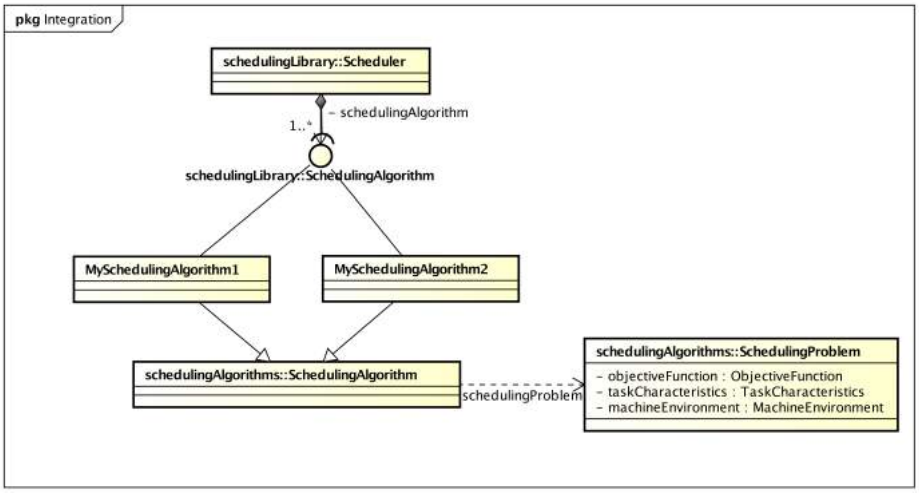


Figure 2.7: Scenario 2: Integrating the scheduling algorithm

this components may add a latency to the scheduling of the subsequent sheets to be printed.

In order to support multiple policies we created two classes which extend class `SchedulingAlgorithm` of the algorithms library. Each of these classes define a different scheduling problem, which should be solved by the corresponding scheduling algorithm.

In this scenario, the difficulty is to access the information needed by the scheduling algorithms and to switch between them. It is possible that we may need to use different scheduling strategies for a batch of sheets due to resource changes. The interesting events that will enable us to choose such sheets may be implicit in the system software. Making these events explicit can increase the number of dependencies and code alterations of the legacy software. This scenario shows the difficulty of non-intrusively selecting objects, thus modularizing them with a criteria other than their type.

2.4 CONCLUSION

This chapter has defined the context of this thesis by referring to similar challenges that are identified in the literature. Then, by referring to an industrial case study we have shown how these problems can arise in systems. In the next two chapters we show our research on how to tackle the problems we have identified in this thesis. Our focus on non-intrusiveness is reflected in our approaches; our aim is to prevent invasive code alterations when integration challenges, like the ones listed above, arise.

Part II

INSTANCE POINTCUTS

In the life-cycle of objects there are different phases. The phase in which an object currently is, affects how it is handled in an application; however, phase shifts are often implicit. Selecting objects according to such phase shifts results in scattered and tangled code. In this part we introduce a new kind of pointcut, called *instance pointcut*, for maintaining sets that contain objects with a specified usage history. Specifics are provided in terms of pointcut-like declarations, selecting events in the life-cycle of objects. Instance pointcuts can be reused, by refining their selection criteria, e.g., by restricting the scope of an existing instance pointcut; and they can be composed, e.g., by set operations. These features make instance pointcuts easy to evolve according to new requirements. Our approach improves modularity by providing a fine-grained mechanism and a declarative syntax to create and maintain phase-specific object sets.

3

INSTANCE POINTCUTS: SYNTAX AND SEMANTICS

In the previous part we have stated the importance of flexibly selecting objects according to their usage history. In this chapter we expand on this problem statement and present our approach which tackles the defined problems.

3.1 INTRODUCTION

In Object-Oriented Programming (OOP), the encapsulated state and the provided behaviour of objects is dictated by their type. Nevertheless, often objects of the same type need to be treated differently. For example, consider a security-enabled system with a type for users. The treatment of a user object depends on the user's privileges and possibly also on the past execution: we may want to reduce the privileges when the user did not change the password for a while, or privileges are added or withdrawn at runtime in other ways.

Software design patterns [GHJV95] are another popular, more general example for dynamically varying the treatment of objects. Several design patterns define *roles* for objects, which can be assigned or removed at runtime, and the roles determine how an object is handled. However, while the pattern localises the handling of object roles, the assignment of roles is usually scattered over multiple source modules. As example, consider the *observer pattern*. To assign the role of *being observed* to a *subject*, an *observer* must be added to its observer list. The code for adding

observers is generally not well localised. Other similar examples are the *adapter*, *decorator*, or *proxy* patterns.

More generally, we can say that objects have a life-cycle and we sometimes need to handle objects according to the life-cycle phase they are currently in. Often the shift from one life-cycle phase to another is implicitly marked by events, e.g., passing an object from one client to another. We claim that to improve the modularity of source code, a declarative definition of relevant object life-cycle phases is necessary. Furthermore, it must be possible to reify the set of objects that currently are in a specified life-cycle phase to consider this information when handling an object. Grouping objects according to criteria which cannot be directly accessed through programming language constructs — such as which class they were initialised in or which method they were passed to as an argument requires invasive insertion of bookkeeping code.

Aspect-Oriented Programming (AOP) can be applied to separate this bookkeeping code from the business logic of the program. But in AOP, *pointcuts* select sets of so-called *join points* which are points in time during the execution of the program. Current aspect-oriented languages do not support a *declarative specification* of the objects belonging to a life-cycle phase; instead an *imperative implementation*, typically following the same pattern, is required for collecting those objects.

A consequence of such an imperative solution is reduced readability and maintainability due to scattering, tangling and boilerplate code. Another issue is the lack of composition and checking mechanisms for the imperative bookkeeping. It is not possible to reuse the previously written code which results in code that is hard to maintain and hinders software evolution. Also the compiler warnings and errors do not indicate the proper context and relevant information to guide the programmer.

To offer better support for processing objects according to their life-cycle phases, we propose a new mechanism, called *instance pointcuts*, to select sets of objects based on the events in their execution history. Instance pointcuts are used to declare the beginning and the end of a life-cycle as events. New instance pointcuts can be defined by reusing

existing ones in two ways: first, by *refining* the expressions defining the relevant events and second, by *composing* two or more instance pointcuts using set operators.

An instance pointcut's concise definition consists of three parts: an identifier, a type which is the upper bound for all the selected objects in the phase-specific set, and a specification of relevant objects. For a basic instance pointcut definition, the specification utilizes *pointcut expressions* to select events that define the start and the end of life-cycle phases and to expose an object. At these events, the object is added to or removed from the set associated with the instance pointcut. We refer to this responsibility as *maintaining* the instance pointcut's object set. New instance pointcuts can be derived from existing ones. Firstly, a new instance pointcut can be derived from another one by restricting the type of selected objects. Secondly, a new instance pointcut can be created by reusing the object selection expressions of the existing ones. Lastly, instance pointcuts can be composed arbitrarily by means of set operators.

In this chapter we present a prototype of instance pointcuts as an extension to AspectJ [KHH+01] and explain its semantics by explaining our compiler which transforms instance pointcuts to plain AspectJ and advanced dispatching library calls.

We reuse the term pointcut for our concept, because it provides a declarative way of specifying crosscuts. Nevertheless, the instance pointcuts select *objects* whose usage crosscuts the program execution rather than *points (or regions) in time* [MEY06] as traditional pointcuts do. Therefore, our instance pointcuts cannot immediately be advised by AspectJ advice, although we offer the possibility to advise the points in time when the extent of instance pointcuts changes (cf. Section 3.4.5.2).

The declarative nature of instance pointcuts enables several compile-time checks which are not automatically possible with equivalent imperative code. Such checks are important to notify the developer when the instance pointcut set is guaranteed to be empty, incompatible types are used in compositions and refinements, etc. These checks help the

developer implement her concern correctly and achieve consistency in phase-specific object sets.

The rest of the part is organised as follows, in Section 3.2 we present a small case study and explain our motivation for the proposed approach, and we formulate a problem statement and goals for our work in Section 3.3. In Section 3.4, a detailed description of instance pointcuts and its various features are presented. Section 3.5 explains how instance pointcuts are compiled. We then present a discussion on the evaluation of our approach in Chapter 4. In the discussion chapter we discuss an application of instance pointcuts for program comprehension and present an evaluation on code quality, performance and check-ability of instance pointcut. We conclude by discussing related work and giving a summary of our approach.

3.2 MOTIVATION

Supporting *unanticipated* extensions or improving already existing code through refactoring may introduce the concern of keeping track of specific objects. Objects can be grouped according to how they are used (passed as arguments to method calls, act as receiver or sender for method calls, etc.) and concerns of an application may be applicable only to objects used in a specific way. Therefore we must be able to identify and select such objects. We want to expose sets of objects belonging to the same life-cycle phase by means of a dedicated language construct such that the implementation of phase-dependent concerns can be explicit.

In Figure 3.1, we outline a part of the architecture of an online shop application. We use this scenario to give examples of grouping objects into sets according to how they are used and how to use these sets in the implementation of concerns.

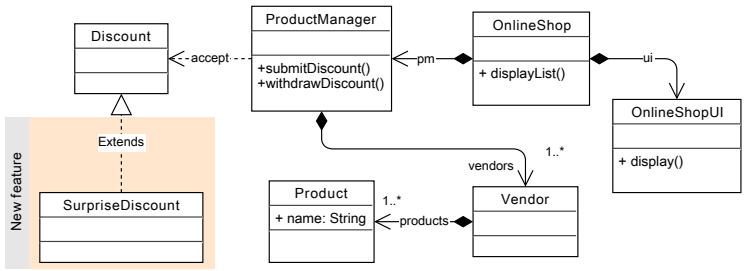


Figure 3.1: Part of an online shop application

3.2.1 *Example Architecture*

In an online shop application, objects of the same type can exist at different phases of their life-cycle. In Figure 3.1 the static structure of a simplified online shop is shown. This structure shows part of the system from the `Vendor` and the `OnlineShop`'s perspective. `Vendors` can submit different kinds of `Discounts` to the `ProductManager` for the `Products` they are selling. The discounts `Vendors` submit are applied to the system on the next day, so there's no option to dynamically declare discounts on products. `Product` is the root of the type hierarchy that represents different kinds of items that are sold in the online shop; i.e., `Product` is parent to classes such as `BeautyProduct` and `SportProduct` (not shown in the figure). Each `Product` holds a list of `Discounts` that are applied to it. The `OnlineShop` has a user interface represented by the `OnlineShopUI` class, which is used to display information to the customers.

3.2.2 *Unanticipated Extensions*

A new feature is added to the online shop which requires creating an alert when a product is applied a surprise discount. The list of surprise discounted products should be available to the user at any time. The surprise discounts are submitted by `Vendors` and they can be submitted or withdrawn any time. In order to realize this extension in an OO-

approach, we need to change several classes. First the class `ProductManager` should keep a set of `Products` to which a surprise discount is applied, in Listing 3.1 this is shown in line 3. This set is updated when a new discount of type `SurpriseDiscount` is submitted or withdrawn (lines 4–19). There can be other components which are interested in this set `surpriseDiscount` being updated. So we need to create a new type of listener, which listens to the changes in this set and handles the change event. In order to add a listener of this type we implement a simple add method `addSurpriseDiscountListener`.

There should also be some changes in the `OnlineShop` class. Since `OnlineShop` is the main application which updates the UI, the changes in the `ProductManager`'s `surpriseDiscount` list is interesting for this class. To listen to these changes it should create a `SurpriseDiscountListener` as an anonymous class (lines 26– 33) and pass this as an argument after instantiating the `ProductManager` in its constructor. It should also include a method for displaying the new UI component; the discount alert which is a small notification that pops up in the UI when a surprise discount is submitted. Once a discount alert is created, an event is fired from the method `updateDiscountAlert` wrapping the created discount alert which is then received by the `updateAndDisplayUI` method. This method checks the type of the event and updates the UI accordingly. To distinguish surprise discount alert events from other events, we should also create a subtype of the class `UIEvent` (line 50) to repaint the relevant components.

```
1  class ProductManager{
2    ...
3    Set<Product> surpriseDiscount = createSet();
4    public void submitDiscount(Product p, Discount d){
5      ...
6      if(d instanceof SurpriseDiscount){
7        surpriseDiscount.add(p);
8        //iteration over sdListenerList
9        listener.handleSurpriseDiscountAdded(p);
10     }
11   }
```

```
12   public boolean withdrawDiscount(Product p, Discount d){
13     ...
14     if(d instanceof SurpriseDiscount){
15       surpriseDiscount.remove(p);
16       //iteration over sdListenerList
17       listener.handleSurpriseDiscountRemoved(p);
18     }
19   }
20   public void addSurpriseDiscountListener(SurpriseDiscountListener
         listener){
21     this.sdListenerList.add(listener);
22   }
23 }
24 class OnlineShop{//SINGLETON
25   ...
26   private SurpriseDiscountListener listener = new
         SurpriseDiscountListener(){
27     public void handleSurpriseDiscountAdded(Product p){
28       updateDiscountAlert(p, true);
29     }
30     public void handle surpriseDiscountRemoved(Product p){
31       updateDiscountAlert(p, false);
32     }
33   }
34   public void init(){
35     ...
36     ProductManager productManager = createProductManager();
37     productManager.addSurpriseDiscountListener(listener);
38   }
39   public void updateDiscountAlert(Product p, boolean display){
40     DiscountAlert discountAlert = //create surprise discount
           alert/remove existing alert for Product p
41     fireListUpdateEvent(new
           SurpriseDiscountUpdateEvent(discountAlert));
42   }
43   public void updateAndDisplayUI(UIUpdateEvent e){
44     //update UI
```

```
45    if(e instanceof SurpriseDiscountUpdateEvent)
46      ui.surpriseDiscountUI().update((SurpriseDiscountUpdateEvent)e);
47   }
48 }

50 class SurpriseDiscountUpdateEvent implements UIEvent{...}
```

Listing 3.1: A Java implementation of discount alert concern

This OO-solution is scattered among the classes `ProductManager` and `OnlineShop` and tangled with multiple methods. It also requires additional classes to be added to the source.

An aspect-oriented implementation can offer a better solution by encapsulating the concern in an aspect. Listing 3.2 shows a possible solution. The set of products which are applied a surprise discount is kept in the aspect (line 2). The following two pointcuts `submit` and `withdraw` selects the products to which a `SurpriseDiscount` is applied (lines 3 – 4). The corresponding advice declarations for these pointcuts maintain the `surpriseDiscount` set. The `submit` pointcut triggers the surprise discount alert method (line 8). There is also the `display` pointcut (line 5), which intercepts the call to `fireListUpdateEvent` method and add the condition for the surprise discount list in an around advice (lines 14 – 19). This aspect includes the implementation of `updateDiscountAlert`, which creates the discount alert and notifies the `OnlineShop`'s UI.

```
1 aspect SDiscount{
2   Set<Item> surpriseDiscount = createSet();
3   pointcut submit(Product p): call(*
        ProductManager.submitDiscount(..)) && args(p,
        SurpriseDiscount);
4   pointcut withdraw(Product p): call(*
        ProductManager.withdrawDiscount(..)) && args(p,
        SurpriseDiscount);
5   pointcut display(UIUpdateEvent event): call(*
        OnlineShop.fireListUpdateEvent(..)) && args(event);
6   after(Product p): submit(p){
7     surpriseDiscount.add(p);
```

```
8     updateDiscountAlert(p, true);
9   }
10  after(Product p):withdraw(p){
11    supriseDiscount.remove(p);
12    updateDiscountAlert(p, false);
13  }
14  void around(UIUpdateEvent event): display(event){
15    if(e instance of SurpriseDiscountUpdateEvent)
16      OnlineShop.instance().ui.surpriseDiscountUI().
17        update((SurpriseDiscountUpdateEvent)e);
18    proceed(event);
19  }
20  public void updateDiscountAlert(Product p, boolean display)
21  {
22    //create/remove discount alert
23    OnlineShop.instance().fireListUpdateEvent(new
          SurpriseDiscountUpdateEvent(discountAlert));
24  }
25  }
```

Listing 3.2: An AspectJ implementation of discount alert concern

3.2.3 *Discussion*

AOP already helps localise the concern and to integrate it to the system with minimal modifications to the existing code. However maintenance of the `surpriseDiscount` set requires the same boilerplate code as the OO solution does. Essentially, the code selects `Product` objects based on the discount they are applied to and deselects them once they are rid of this discount. This marks a phase in the life-cycle of a `Product` object. Traditional AOP solutions also fail to declare the *life-cycle concern* in a declarative manner; although AOP localises the concern into a single compilation unit, the concern is still implemented as separate imperative statements. Furthermore, reusing such existing reifications of objects in a specific life-cycle phase by refining or composing them is not conve-

niently supported at all; For example, if we want to find the subset of `BeautyProducts` of the `surpriseDiscount` set, we have to iterate over it and check instance types to create a new set. Such imperative definitions are difficult or impossible to analyse by the compiler. For instance, it may be desirable to warn developers about pointcuts that probably will never match any object, e.g., because the selection events will never happen. With a declarative notation, a compiler would be able to identify such situations. These observations led us to our problem statement which is laid out in the next section.

3.3 PROBLEM STATEMENT

In the previous section we have demonstrated two things: First, the implementation of some concerns requires accessing groups of objects with similar usage history. Second, making such groups accessible to the program in a modular and re-usable way is not supported by current programming languages.

Since creating object sets according to execution events is a crosscutting concern, we claim that a new programming technique in the style of aspect-oriented programming is required for modularising concerns depending on object groups. Such a programming construct must satisfy the following needs:

1. A declarative way of selecting/de-selecting objects according to the events they participate in should be provided.
2. The selected objects should be kept in a data structure that does not allow duplicates.
3. The set of objects should be accessible and any changes to this set, i.e., adding/removing objects, should create a notification.
4. For the same kind of objects, the sets should be composable to obtain new sets and the composition should also satisfy the above requirements.

Our first requirement is derived from our main observation about the need for selecting objects based on their usage history. The second requirement is due to the fact that the same object can participate in the same event more than once; in this case we do not need to add this object to the group over and over again. Counting the number of times an object participates in an event can be relevant. Therefore a set data structure where each element has a cardinality, defined as the number of times it was added to the set, should be used. The third requirement has to do with a new modularisation mechanism. As we have mentioned in the motivation, localisation and modularisation are not always the same thing. A concern can be localised in a single compilation unit meaning its implementation can be in a single file. However this does not make sure that the concern is implemented in a modular way. In fact as we have shown in the AspectJ implementation of our example, the actual concern of adding and removing surprise discount is implemented in a fragmented way. This implementation is neither reusable nor extensible. The fourth requirement is also related to the imperative programming of collections in programming languages.

3.4 INSTANCE POINTCUTS

To support the requirements outlined in the previous section, we propose a new kind of pointcut for declaratively selecting objects based on their life-cycle phases, where the beginning and the end of a phase is marked by events. An instance pointcut is a declarative language construct that is used to reify and maintain a set of objects of a specified type. The objects are selected over a period marked by events in their life-cycle. Instance pointcuts modularise the object selection concern and make it declarative.

In the remainder of this section, we explain instance pointcuts in detail. The concept of instance pointcuts is language-independent; it can be implemented as an extension to arbitrary OO-based aspect-oriented languages. In this work, we have implemented a prototype as an extension

to AspectJ. Since we inherit AspectJ's join-point model, our prototype also shares some of its limitations. Nevertheless the AspectJ extension provides a good illustration to how instance pointcuts can be used. In this section we provide many examples in our AspectJ-based instance pointcuts implementation.

3.4.1 *Basic Structure and Properties*

A concrete instance pointcut definition consists of a left hand-side and a right-hand side (Figure 3.2, rule 1). At the left-hand side the pointcut's name and a type is declared. Each instance pointcut declaration starts with a `static` modifier followed by the keywords `instance pointcut`. An instance pointcut does not declare pointcut parameters since it has the specific purpose of exposing one object from an event; it has a single implicit parameter called `instance` of the declared type.

At the right-hand side the instance pointcut expression selects the desired events from join points and then binds the exposed object (represented by the `instance` parameter) as a member of the instance pointcut's set.

⟨*instance pointcut*⟩ ::= '`instance pointcut`' ⟨*name*⟩ '`<`' ⟨*instance-type*⟩ '`>`' '`:`'
⟨*ip-expr*⟩ ('`UNTIL`' ⟨*ip-expr*⟩)?

⟨*ip-expr*⟩ ::= ⟨*after-event*⟩ '`||`' ⟨*before-event*⟩
| ⟨*before-event*⟩ '`||`' ⟨*after-event*⟩
| ⟨*after-event*⟩
| ⟨*before-event*⟩

⟨*after-event*⟩ ::= '`after`' '`(`'⟨*pointcut-expression*⟩'`)`'

⟨*before-event*⟩ ::= '`before`' '`(`'⟨*pointcut-expression*⟩'`)`'

Figure 3.2: Grammar definition for instance pointcuts

3.4.2 *Add/Remove Expressions*

An instance pointcut expression is a composition of two *sub-expressions* separated by the `UNTIL` keyword: 1. The add expression which selects the events at which objects are added to an instance pointcut's set, is mandatory. 2. The remove expression which selects the events at which objects are removed from the set, is optional. The 'add to set' and 'remove from set' operations are implicitly performed when certain events specified in the corresponding sub-expression (cf. Figure 3.2, rule 1) occur. In natural language we can describe the semantics of an instance pointcut as: when an add event occurs add the instance of the desired type into a set *until* a remove event occurs in which the same instance participates in.

In AspectJ, join points mark *sites* of execution; a join point by itself does not define an event. Pointcut expressions select join points and pointcuts are used with advice specifications to select a particular event in that join point. As discussed by Masuhara et al. [MEY06] such a region-in-time join-point model hinders re-use of pointcuts.

In our prototype we combine pointcut expressions with advice specifiers and obtain *expression elements*. Each expression element contains a pointcut expression, which matches a set of join points. Then, from these join points, according to the advice specifier the before or after events are selected. Both add and remove expressions are composed of expression elements which can be a *before element* or an *after element* (Figure 3.2, rule 3-4). A sub-expression (add/remove expression) contains at least one *expression element* and at most two. In Figure 3.2 the second grammar rule depicts this statement.

In Figure 3.2 rules 3 and 4 contain the $\langle pointcut - expression \rangle$ rule which represents an AspectJ pointcut expression. However we have introduced a restriction that in every pointcut expression, there must be *exactly* one binding predicate (`args`, `target` etc.) that binds the `instance` parameter. Furthermore, it is mandatory to bind the `instance` parameter, since it represents the object to be added or removed from the set.

Allowing only the binding of one value at each event is a limitation of our current language prototype. It would be a straight-forward extension of the instance pointcut language to allow binding multiple values to the implicit `instance` parameter and then add all bound values to the instance pointcut's set (e.g the passed parameter with args and the returned object with `returning`). However, this would require a more complex code generation.

The binding predicates are extended to include the `returning` clause. The `returning` clause binds the value returned by a method or a constructor. In AspectJ the syntax is restricted and `returning` can only be used in an after advice, since the returned value is only available after a method finishes execution, this is also true in our case. Although we do not have this restriction syntactically, we enforce that the `returning` clause is used only with the after event selector by means of a semantic check.

In an instance pointcut expression, it is only possible to *OR* a before event with an after event. The *before* clause selects the start of executing an operation (i.e., the start of a join point in AspectJ terminology) and the *after* clause selects the end of such an execution. For two operations that are executed sequentially, the end of the first and the start of the second operation are treated as two different events. Thus, the before and after clauses select from two disjoint groups of events and the conjunction of a before and an after clause will always be empty.

```
1 static instance pointcut surpriseDiscount<Product>:
2   after(call(* ProductManager.submitDiscount(..))
3   && args(instance, SurpriseDiscount))
4   UNTIL
5   after(call(* ProductManager.withdrawDiscount(..))
6   && args(instance, SurpriseDiscount));
```

Listing 3.3: A basic instance pointcut declaration with add and remove expressions

The instance pointcut in Listing 3.3 shows a basic example of the instance pointcut that defines the same behavior as was discussed in our motivating example in Section 3.2. The left-hand side of the instance

pointcut indicates that the pointcut is called `surpriseDiscount` and it is interested in selecting `Product` objects. On the right hand side, there are two expressions separated by the `UNTIL` keyword. The first one is the add expression. It selects the join-point marked by the method `submitDiscount` and from the context of this event it exposes the `Product` object with the `args` clause and binds it to the `instance` parameter. The second one is the remove expression and it selects the after event `withdrawDiscount` call and exposes the `Product` instance in the method arguments and binds it to the `instance`.

Note that instance pointcuts do not keep objects alive, as instance pointcuts are non-invasive constructs, which do not affect the program execution in any way. So even if the remove expression was not defined for the `surpriseDiscount` instance pointcut, when the `Product` instances are collected by the garbage collector, they are removed from the set.

3.4.3 *Multisets*

An instance pointcut reifies an object set as a *multiset*. A multiset, also referred to as a *bag*, allows multiple appearances of an object. Every contained object has a corresponding cardinality which indicates its multiplicity in the set.

The instance pointcut shown in Listing 3.4 selects `Product` instances, which are applied a `Discount`. The remove expression removes a `Product` instance if the `Discount` is removed from that `Product`. With this pointcut we would like to represent the currently discounted products. Multiset makes sure that `Products` can be added for each discount submission operation. When the same product is added with different types of discounts, and if one of the discounts is removed, then still one entry of that instance is left in the set. If instance pointcuts only supported a set then as soon as a discount is removed from a product, its only copy would be removed and it would appear as if there are no more discounts on that product.

```
1 static instance pointcut multi_discount<Product>:
```

```
after(call(* ProductManager.submitDiscount(..))
&& args(instance))
UNTIL
after(call(* ProductManager.withdrawDiscount(..))
&& args(instance));
```

Listing 3.4: An instance pointcut utilizing multiset property

3.4.4 *Refinement and Composition*

Instance pointcuts can be referenced by other instance pointcuts. They can be refined in two ways and they can be composed together to create new instance pointcuts.

3.4.4.1 *Referencing and Type Refinement*

Instance pointcuts are referenced by their names. Optionally the reference can also take an additional statement for *type refinement*, which selects a subset of the instance pointcut that is of the specified type. Type refinements require that the refinement type is a subtype of the original instance type. For example, the instance pointcut `surpriseDiscount` (Listing 3.3) can be refined as shown in Listing 3.5. The refinement expression selects the subset of `BeautyProduct` instances from the set of `Product` instances selected by the `surpriseDiscount` instance pointcut. The `surpriseDiscountBeauty` instance pointcut is defined using the result of this expression. Note that with this notation objects that are of a subtype of `BeautyProduct` will also be selected (equivalent to an `instanceof` check).

```
static instance pointcut surpriseDiscountBeauty<BeautyProduct>:
 surpriseDiscount<BeautyProduct>;
```

Listing 3.5: A type refined pointcut

3.4.4.2 *Instance Pointcut Expression Refinement*

In Section 3.4.2 we have introduced the instance pointcut expression, which consists of two sub-expressions (add and remove expressions). We provide an expression refinement mechanism which makes it possible to reuse parts of the existing instance pointcut expressions to create new ones. The expression elements forming the sub-expressions can be accessed individually to be extended by concatenating other primitive pointcuts, so-called *refinement expressions*, with boolean operators. We offer a naming convention to access parts of the instance pointcut expression with different granularity. Note that this syntax is only valid when used in the context of an expression refinement.

<IP-REF> When an instance pointcut is referenced directly then the refinement expression is composed with the pointcut expression in all of the before and after event selectors, in the add and remove expressions.

<IP-REF>.{ADD, REMOVE} This expression provides access at the sub-expression level. The refinement expression is composed with the pointcut expressions in referenced sub-expression's before and after event selectors.

<IP-REF>.{ADD, REMOVE}_{AFTER, BEFORE} This naming convention is used to access the pointcut expressions of the individual before and after event selectors and provides the finest granularity. In fact, the other two access statements can be written in terms of this one, since they just provide a short hand for the collective expression refinements.

It is possible to compose any primitive pointcut, except the binding predicates, with a sub-expression. Although we chose not to restrict this aspect, some compositions will not be meaningful for selecting objects. For example, composing an `execution` pointcut with an expression that already includes a `call` pointcut will result in a non-matching pointcut expression. This is further discussed in Section 4.2.

Let us explain the usage of the expression access by examples. The example shown in Listing 3.6 shows a reuse of the `surpriseDiscount`'s sub-expressions to create a new instance pointcut. The newly created pointcut's sub-expression can be accessed through the aforementioned naming conventions.

```
1 static instance pointcut surpriseDiscountOver50<Product>:
2   surpriseDiscount.add && if(instance.getPrice() > 50) UNTIL
3     surpriseDiscount.remove;
```

Listing 3.6: Expression refinement of `surpriseDiscount` (Listing 3.3) instance pointcut

The `if` pointcut in Listing 3.6 is appended to the add expression of the instance pointcut `surpriseDiscount` (Listing 3.3). The effect of this composition is as follows; the `if` pointcut will be appended to all of the pointcut expressions contained in the `after` and `before` event selectors. Since the `surpriseDiscount` pointcut only has one after event in its add expression, the resulting add expression is equivalent to the expression shown in Listing 3.7.

```
after(call(* ProductManager.submitDiscount(..)) &&
args(instance, SurpriseDiscount) && if(instance.getPrice() > 50))
```

Listing 3.7: Equivalent add expression of the expression refinement shown in Listing 3.6

Expression refinements can also be used for more precise type refinements. Revisiting the example given in Section 3.4.4.1, the `surpriseDiscountBeauty` instance pointcut (Listing 3.5) can be constructed to include instances with the *exact type* `BeautyProduct` (Listing 3.8). The effect is different from type refinement since `surpriseDiscountOnlyBeauty` does not include subtypes of `BeautyProduct`.

```
1 static instance pointcut surpriseDiscountOnlyBeauty<BeautyProduct>:
2   surpriseDiscount &&
        if(instance.getClass().equals(BeautyProduct.class));
```

Listing 3.8: Type refinement by expression refinement

3.4.4.3 *Instance Pointcut Composition*

Instance pointcuts reify sets, for this reason we facilitate the composition in terms of the set operations *intersection* and *union*. In Figure 3.3, an extended version of the grammar definition is shown. The composition of two instance pointcuts creates a *composite* instance pointcut. Different from regular instance pointcuts, composite ones are declared with the keyword `composite` and they do not have instance pointcut expressions. Instead they monitor the component instance pointcuts' set change operations and update their own set accordingly. In order to declare a set intersection the keyword `inter` and to declare a set union the keyword `union` is used. Throughout the text we use the mathematical symbols for these operations, ∩ as intersection and ∪ as union. Since composite instance pointcuts do not have an instance pointcut expression they cannot be used in expression refinement. However, they can be type-refined; the result of the type refinement of a composite instance pointcut is also a composite instance pointcut and must be declared as such.

The type of a composite instance pointcut must be assignment compatible to the types of the component instance pointcuts. It is also possible to leave out the type declaration and let the compiler infer the type. For a composition of two instance pointcuts, the type of the composite one can be determined depending on the relation of the types of the component instance pointcuts. For illustration of this type inference, consider the type hierarchy in Figure 3.4a: `R` is the root of the hierarchy with the direct children `A` and `B` (i.e., these types are siblings); `C` is a child of `B`. Figure 3.4b shows four distinct cases: Either the type of one of the instance pointcuts is a super type of the other one's type (second row), or both types are unrelated (third row); and the composition can either be ∩ (third column) or ∪ (fourth column).

When composing two instance pointcuts with types from the same hierarchy, the type of the composition is the more specific type (`C` in the example) for an ∩ composition and the more general type (`B`) for an ∪ composition. When composing two instance pointcuts with sibling types, for the ∩ operation the resulting composition cannot select any types

⟨*instance-pointcut*⟩ ::= '`composite instance pointcut`' ⟨*name*⟩ ('<' ⟨*instance-type*⟩ '>')? ':' ...
 | ⟨*comp-expr*⟩

⟨*comp-expr*⟩ ::= ⟨*comp-expr*⟩ '`inter`' ⟨*comp-expr-t*⟩
 | ⟨*comp-expr-t*⟩

⟨*comp-expr-t*⟩ ::= ⟨*comp-expr-t*⟩ '`union`' ⟨*comp-expr-f*⟩
 | ⟨*comp-expr-f*⟩

⟨*comp-expr-f*⟩ ::= ⟨*ip-ref*⟩
 | '(' ⟨*comp-expr*⟩ ')'

⟨*ip-ref*⟩ ::= ⟨*name*⟩
 | ⟨*name*⟩('<' ⟨*refined-instance-type*⟩ '>')?

Figure 3.3: Syntax for instance pointcut composition

since the types `A` and `B` cannot have a common instance. The $\cup$ operation again selects a mix of instances of type `A` and `B`, thus the composed instance pointcut must have the common super type, `R` in the example.

Because instance pointcuts are reified as multisets, these operations are different from the regular set operations. The definition of the intersection and union operations for multisets is given in the next definition.

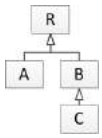
Definition 1 *Assume* (X, f) *and* (Y, g) *are multisets, where* X, Y *represents the elements and* f, g *represents a function which maps each element to a cardinal number.*

The ***intersection*** *of these sets is defined as* (V, h) *where,*

$$V = X \cap Y$$

and $\forall v \in V$ *the multiplicity of* v *is defined as*

$$h(v) = \min(f(v), g(v))$$



(a) A simple type hierarchy

pc1 type	pc2 type	∩	∪
B	C	C	B
B	A	∅	R

(b) Instance pointcut compositions and effect on the captured instance type

Figure 3.4: An example to illustrate composition's effect on types

*The **union** of these sets is defined as* (Z, i) *where,*

$$Z = X \cup Y$$

and $\forall z \in Z$ *the multiplicity of* z *is defined as*

$$i(z) = max(f(z), g(z))$$

3.4.5 *Using Instance Pointcuts*

Up to now we have explained the syntax and semantics for definitions of instance pointcuts. In this section we explain how to use instance pointcut in the context of an AO language, namely, AspectJ. As example, throughout this section, we use the instance pointcut defined in Listing 3.9, which maintains a set of `Products` that are currently out of stock. Instance pointcuts are static members of classes and can have any visibility modifier. Thus, all modules, aspects as well as classes, that can see an instance pointcut can use it in the ways described below.

```
1 static instance pointcut outOfStock<Product>:
2   after(call(* Product.outOfStock(..)) &&
```

```
3         target(instance))
4     UNTIL
5     after(call(* Vendor.stock(..))
6         && args(instance));
```

Listing 3.9: An instance pointcut for out of stock products

3.4.5.1 *Set Access*

Instance pointcuts reify a phase-specific object set and this set can be accessed through a static method, which has the same name as the instance pointcut identifier. Only the *get* methods of the collection interface can be used to retrieve objects from the set. Write methods, which modify the contents of the set, are not allowed since they create data inconsistencies like adding an object which is not in the same life-cycle phase as the ones selected by the instance pointcut. We ensure this by returning an `UnmodifiableSet` from the set access methods. In Listing 3.10 the `outOfStock()` method (line 4) returns the set of `Products` that are currently out of stock.

```
1 public static double calculateDamages()
2 {
3   double damage = 0;
4   for(Product p: MyAspect.outOfStock())
5     damage = damage + p.getPrice();
6   return damage;
7 }
```

Listing 3.10: Calculate a damage estimate for out of stock products

3.4.5.2 *Set Monitoring*

An instance pointcut definition provides two set change events, an add event and a remove event. In order to select the join points of these events, every instance pointcut definition automatically has two implicit regular pointcuts. These implicit pointcuts have the following naming conventions, ⟨name⟩_instanceAdded, ⟨name⟩_instanceRemoved, where

⟨name⟩ is the name of the instance pointcut. In Listing 3.11, a before advice using the `outOfStock_instanceAdded` pointcut is shown. When a product is marked out of stock and it is added to the set, a notification is sent to the related `Vendor` indicating that the product is out of stock.

```
1 before(Product p): outOfStock_instanceAdded(p)
2 {
3   OnlineShop.notifyVendor(p.getVendor, STOCK_MSG);
4 }
```

Listing 3.11: Set monitoring pointcut used to notify vendors

3.5 COMPILATION OF INSTANCE POINTCUTS

A goal for our compiler implementation is to support modular compilation. This means to compile an aspect with instance pointcuts that refer to instance pointcuts defined in other aspects, it must be sufficient to know their declaration (i.e., the name and type); it should not be necessary for the compiler to know the actual expression of the referenced instance pointcuts.

We have implemented the instance pointcut language using code transformation employing two tools. First, the parser of our language and the code generation templates are implemented with the EMFText[1] language workbench. For this purpose, we have defined the AspectJ grammar by using JaMoPP[2] [HJSW10] as the foundation and extended it with the grammar for instance pointcuts which was presented interspersed with the previous section.

Second, the generated code uses the ALIA4J[3] [BSY+12] framework for so-called advanced dispatching language implementations. The term advanced dispatching refers to late-binding mechanisms including, e.g., predicate dispatching and pointcut-advice mechanisms. At its core, ALIA4J

1 EMFText, see http://www.emftext.org/
2 JaMoPP: Java Model Parser and Printer, see http://jamopp.inf.tu-dresden.de
3 The Advanced-dispatching Language Implementation Architecture for Java. See http://www.alia4j.org/alia4j/.

contains a meta-model of advanced dispatching declarations in which AspectJ pointcut and advice, as well as instance pointcuts can be expressed.

We use ALIA4J to realise the crosscutting behaviour of our language instead of AspectJ because the way AspectJ handles binding of values and restricting their types in pointcuts would prohibit a modular compilation of instance pointcuts. While in instance pointcuts value binding is uniformly expressed in a pointcut expression, in AspectJ binding the result value must be specified in the advice definition (via the `after returning` keyword) and all other values are bound in pointcut expressions. Therefore, AspectJ code generated for an instance pointcut expression would have to depend on which value is bound; this means that the code generation for a derived instance pointcut would also depend on the binding predicate (an implementation detail) of the referenced one. It is not possible to work around this using AspectJ's reflective `thisJoinPoint` keyword, as it does not expose the result value at all. Another, similar limitation is that AspectJ does not allow to narrow down the type restriction for the bound value of a referred pointcut. Thus, in order to be able to transform an instance pointcut with type refinement to AspectJ, it is necessary to know the definitions of the referenced instance pointcuts and inline them.

Our compiler generates different code depending on whether the instance pointcut is a composite one or not and whether it is a refinement of an instance pointcut or not. Common to all cases is the code for managing the data of the instance pointcut. Listing 3.12 exemplary shows that code; the variables *${Type}* and *${ipc}* stand for the instance pointcut's type and name, respectively. The natural text written in the comments provides a description of the code for which it stands.

First, to store the instances currently selected by an instance pointcut as a multiset, a `WeakHashMap` is defined (cf. line 1); the keys of the map are the selected objects and the mapped value is the cardinality. We use weak references to avoid keeping objects alive which are not reachable

from the base application anymore. The generated method *${ipc}* returns all objects which are currently mapped (cf. lines 2–4).

Methods are also generated to access, increase or decrease the counter of selected objects; if an object does not have an associated counter yet, or the counter reached zero, the object is added to or removed from the map, respectively (cf. lines 5–15). After having performed their operations, `${ipc}_addInstance` and `${ipc}_removeInstance` methods invoke an empty method, passing the added or removed object. We generate a public, named pointcut selecting these calls, exposing the respective events (cf. lines 18 and 19).

```
1  private static WeakHashMap<${Type}, Integer> ${ipc}_data = new
       WeakHashMap<${Type}, Integer>();
2  public static Set<${Type}> ${ipc}() {
3   return Collections.unmodifiableSet(${ipc}_data.keySet());
4  }
5  public static void ${ipc}_addInstance(${Type} instance) {
6   //increase counter associated with instance by the ${ipc}_data map
7   ${ipc}_instanceAdded(instance);
8  }
9  public static void ${ipc}_removeInstance(${Type} instance) {
10  //decrease counter associated with instance by the ${ipc}_data map
11  //if the counter reaches 0, remove instance from the map
12  ${ipc}_instanceRemoved(instance);
13 }
14 public static int ${ipc}_cardinality(${Type} o) {..}
15 private static void ${ipc}_setCardinality(${Type} o, int c){..}
16 private static void ${ipc}_instanceAdded(${Type} instance) {}
17 private static void ${ipc}_instanceRemoved(${Type} instance) {}
18 public pointcut ${ipc}_instanceAdded(${Type} instance) :
       call(private static void Aspect.${ipc}_instanceAdded(${Type}))
       && args(instance);
19 public pointcut ${ipc}_instanceRemoved(${Type} instance) :
       call(private static void
       Aspect.${ipc}_instanceRemoved(${Type})) && args(instance);
```

Listing 3.12: Template of generated code for instance set management.

Next, these bookkeeping methods have to be executed at events corresponding to the instance pointcut definitions. Below, we elaborate on the code generation for instance pointcuts defined in the different possible ways.

3.5.1 *Non-Composite Instance Pointcuts*

A non-composite instance pointcut, generally consists of four underlying pointcut definitions: specifying join points (1) *before* or (2) *after* which an instance is to be *added* to the selected instances; and specifying join points (3) *before* or (4) *after* which an instance is to be *removed*. For each pointcut definition, we generate a method that creates a corresponding advanced dispatching model; the methods are called `${ipc}_add_before`, `${ipc}_add_after`, `${ipc}_remove_before`, and `${ipc}_remove_after`.

In the meta-model, a *Specialization* can represent a partial AspectJ pointcut and a full pointcut expression can be represented as the disjunction of a set of *Specializations* (discussed in detail elsewhere [BM07]). Figure 3.5 shows the meta-model for a Specialization in ALIA4J consisting of three parts. A *Pattern* specifies syntactic and lexical properties of matched join point shadows. The *Predicate* and *Atomic Predicate* entities model conditions on the dynamic state pointcut designators depend on. The *Context* entities model access to values like the called object or argument values. Contexts which are directly referred to by the Specialization are exposed to associated advice (i.e., they represent binding predicates).

Depending on the definition of the instance pointcut, advanced dispatching models of the underlying pointcuts have to be created in different ways. All four underlying pointcuts are optional; a missing pointcut can be represented as an empty set of Specializations in the meta-model.

PLAIN INSTANCE POINTCUTS For pointcut expressions that are directly provided, we use a library function provided by ALIA4J which

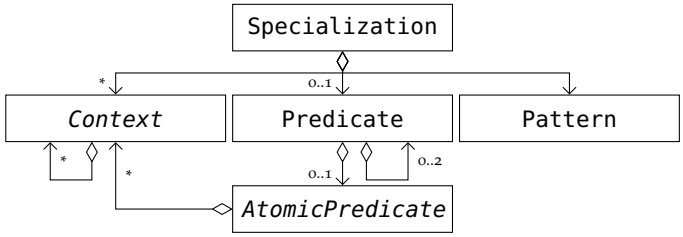


Figure 3.5: Meta-model of a Specialization in ALIA4J.

takes a String containing an AspectJ pointcut as input. We have extended this library to also accept the `returning` pointcut designator.

```
1 static instance pointcut ${ipc}<${Type}>:
2 before(${pc_add_before}) after(${pc_add_after}) UNTIL
3 before(${pc_remove_before}) after(${pc_remove_after};
```

Listing 3.13: Example of a plain instance pointcut

For the example instance pointcut presented in Listing 3.13, we show the code generated for the method creating the advanced dispatching model for the add_before pointcut in Listing 3.14; the other methods are generated analogously. Line 4 shows the transformation of an AspectJ pointcut into a set of Specializations in the meta-model by passing the pointcut as a String—represented by *${pc_add_before}* in Listings 3.13 and 3.14—to the above mentioned library function.

```
1 private static Set<Specialization> ${ipc}_add_before;
2 public static Set<Specialization> ${ipc}_add_before() {
3   if (${ipc}_add_before == null) {
4     ${ipc}_add_before = Util.toSpecializations("${pc_add_before}",
          ${Type});
5   }
6   return ${ipc}_add_before;
7 }
```

Listing 3.14: Template for creating the advanced dispatching model for the add_before expression

TYPE REFINEMENT An instance pointcut can also be defined by referring to another instance pointcut whereby a type restriction for the selected instances can be defined. A template for an instance pointcut defined in such a way is as follows:

```
static instance pointcut ${ipc}<${Type}>: ${ipc1}<${Type}>;
```

For such an instance pointcut methods are also created to produce advanced dispatching models for the four underlying pointcuts (see the template in Listing 3.15). These methods first invoke the corresponding methods of the referenced instance pointcut (cf. line 3). Second, the type restriction is added to the Predicates of the retrieved Specializations (cf. line 4) with the method `addTypeConstraint`.

```
1 public static Set<Specialization> ${ipc}_add_before() {
2   ...
3   Set<Specialization> ipRef = ${ipc1}_add_before();
4   ${ipc}_add_before = Util.addTypeConstraint(ipRef, ${Type});
5   ...
6 }
```

Listing 3.15: Template for creating the advanced dispatching model for the type-refined instance pointcut

EXPRESSION REFINEMENT When defining a new instance pointcut through expression refinement, for each of the four underlying pointcut expressions, a plain pointcut expression can be *and*ed or *or*ed with the underlying pointcut expression of the referenced instance pointcut. As explained in Section 3.4, refinement pointcut expressions must not include a binding predicate. The referred instance pointcut expression already has a binding predicate which is carried over to be used as the binding predicate for the newly composed pointcut expression. The template for the generated method for creating the add_before advanced dispatching model is shown in Listing 3.16. It assumes that the instance pointcut is named `${ipc}` and it refines another instance pointcut `${ipc1}` by *and*ing the pointcut expression `${ipc1_add_before}` to the add/be-

fore underlying pointcut. If the pointcuts are *or*ed, correspondingly the method `orSpecializations` is used in line 5.

These utility methods are provided as runtime library for the instance pointcuts. The method `andSpecializations` forms the conjunction of the Predicates and Patterns of the passed Specialization sets. If `ipRef` is empty, the conjunction is also empty and an empty set is returned. Otherwise, the Context declared by the Specializations `ipRef` is copied to the ones newly created by the `andSpecializations` method. The method `orSpecializations` is implemented similarly; but it forms the disjunction of Predicates and Patterns and if `ipRef` is empty, an exception is raised.

```
1 public static Set<Specialization> ${ipc}_add_before() {
2   ...
3   Set<Specialization> plainIPExpr =
        Util.toSpecializatons("${pc_add_before}", ${Type});
4   Set<Specialization> ipRef = ${ipc1}_add_before();
5   ${ipc}_add_before = Util.andSpecializations(ipRef, plainIPExpr);
6   ...
7 }
```

Listing 3.16: Generated code for creating the advanced dispatching model for the add/before pointcut of the instance pointcut created with expression refinement

DEPLOYMENT In each of the above cases, the created advanced dispatching models of the pointcuts must be associated with advice invoking the add or remove method for the instance pointcut. In a advanced dispatching model this is achieved by defining an *Attachment*, which roughly corresponds to a pointcut-advice pair. An Attachment refers to a set of Specializations, to an *Action*, which specifies the advice functionality, and to a *Schedule Information*, which models the time relative to a join point when the action should be executed, e.g., "before" the join point (cf. line 6).

Listing 3.17 shows the generated code for creating and deploying the bookkeeping Attachments. The first Attachment uses the set of Special-

izations returned by the *${ipc}*add_before method (cf. line 4) and specifies the ${ipc}_addInstance method as action to execute at the selected join points (cf. line 5). As relative execution time, the Attachment uses a "SystemScheduleInfo"; this is provided by ALIA4J for Attachments performing maintenance whose action should be performed before or after all user actions at a join point, such that all user actions observe the same state of the maintained data. Thus, when reaching a selected join point the instance is added to the instance pointcut's multiset before any other action can access its current content. The other Attachments are created analogously. In the end, all Attachments are deployed through the ALIA4J System (cf. line 2).

```
1  public static void ${ipc}_deploy() {
2   org.alia4j.fial.System.deploy(
3     new Attachment(
4       ${ipc}_add_before(),
5       createStaticAction(void.class, ${Aspect}.class,
            "${ipc}_addInstance", new Class[]{${Type}.class})
6       SystemScheduleInfo.BEFORE_FARTHEST),
7     //Create Attachments for the other three parts analogously.
8     //For the ''after'' parts, use SystemScheduleInfo.AFTER_FARTHEST.
9     //For the ''remove'' parts, specify method ${ipc}_removeInstance.
10    );
11 }
```

Listing 3.17: Deployment of the bookkeeping for an instance pointcut.

3.5.2 *Composite Instance Pointcuts*

Composite instance pointcuts require a different compilation strategy because they do not have the four underlying pointcut expressions. The data of a composite instance pointcut changes when the data of one of its referenced instance pointcuts is updated. The corresponding events happen during the execution of the generated methods *${ipc}*_addInstance and *${ipc}*_removeInstance. Therefore, a different mechanism is needed

than for the non-composite instance pointcuts which depend on user events.

INTERSECTION AND UNION When an instance pointcut *${ipc}* is composed by forming the union or intersection of other instance pointcuts (*${ipcX}*), the content of the maintained multiset potentially changes whenever an instance is added to or removed from one of the referenced instance pointcuts, events already exposed through *${ipcX}*_instanceAdded or *${ipcX}*_instanceRemoved pointcuts. For the maintenance of a composite instance pointcut a method is generated which reacts to the join points matching the disjunction of all these pointcuts. The argument of this method is the instance exposed by these pointcuts, i.e., the instance that has either been added to or removed from a reference instance pointcut. When the maintenance method is invoked, we know the cardinality of this instance potentially changes in the multiset of the instance pointcut *${ipc}*. The cardinality of other instances cannot change. The generated method, therefore, re-calculates the cardinality of the affected instance and changes its value in *${ipc}*_data.

To generate appropriate code, the compiler first builds a binary expression tree for the composition expression. Next, it traverses this tree and generates different code for the cases that the visited node is an instance pointcut reference, or an `inter` or `union` operator. For an instance pointcut reference, code is generated that retrieves the cardinality of the instance in the multiset of the referenced instance pointcut. For an `inter` and `union` operator, code is generated that calculates the minimum and maximum, respectively, of both sub-expressions. Finally, the cardinality in *${ipc}*_data is updated.

As example, Listing 3.18 shows the generated code for a composite instance pointcut with the set expression (*${ipc1}* **union** *${ipc2}*)**inter** *${ipc3}*. Besides, the generated code remembers the old cardinality; when the cardinality changes from 0 to > 0 or vice versa, the generated method invokes the method *${ipc}*_instanceAdded *${ipc}*_instanceRemoved, respectively.

```
public static void ${ipc}_update(Object o) {
  int oldCardinality = ${ipc}_cardinality(o);
  int newCardinality =
    Math.min(
      Math.max(
        ${ipc1}_cardinality(o),
        ${ipc2}_cardinality(o)),
      ${ipc3}_cardinality(o));
  ${ipc}setCardinality(o, newCardinality);
  if (oldCardinality == 0 and newCardinality > 0)
    ${ipc}_instanceAdded(o);
  else if (oldCardinality > 0 and newCardinality == 0)
    ${ipc}_instanceRemoved(o);
}
```

Listing 3.18: The update method generated from a composition expression

As in the case of non-composite instance pointcuts, an advanced dispatching meta-model Attachment is generated and deployed which associates the Specializations corresponding to the pointcuts with the generated method.

TYPE REFINEMENT OF COMPOSITE INSTANCE POINTCUTS Instance pointcuts which are defined by means of type refined composite instance pointcuts, are treated similar to the case above. A method is generated which is executed when the referenced instance pointcut changes. The method checks whether the type of the added or removed object is assignment compatible with the type restriction. If this is the case, the same operation (adding or removing the instance) is performed on the multiset of the refining instance pointcut.

3.5.3 *Compiling Plain AspectJ constructs*

To ensure consistent ordering between AspectJ advice and our implementation of instance pointcuts (i.e., that our bookkeeping advice are executed before user advice), the AspectJ pointcut-advice definitions

must be processed by ALIA4J. This is possible because ALIA4J can integrate with the standard AspectJ tooling. Using command line arguments the AspectJ compiler can be instructed to omit the weaving phase. The advice bodies are converted to methods and pointcut expressions are attached to them using Java annotations which are read by the ALIA4J-AspectJ integration and transformed into Attachments at program start-up. The code generated by our compiler consists of the above explained methods, as well as plain AspectJ definitions. When compiling this code with the mentioned command line options, the regular AspectJ pointcut-advice and the behaviour of the instance pointcuts are both executed by ALIA4J, thus ensuring a consistent execution order.

4

INSTANCE POINTCUTS: DISCUSSION

The instance pointcuts approach satisfies the goals we have stated at the beginning of Section 3.4 and in Section 3.5 we have shown that instance pointcuts can be compiled modularly. Our approach provides:

- A concise syntax, which is used to generate the necessary bookkeeping code,
- additional features to declaratively create refined sets, reusing already created instance pointcuts and,
- a composition mechanism, which uses set operations and allows modular definition of instance pointcuts.

In this chapter we discuss an application of instance pointcuts in the program comprehension domain (Section 4.1). This is followed by an evaluation which includes discussions about code quality improvements, performance characteristics and enabled analyses (Section 4.2). We conclude this chapter by presenting related work (Section 4.3) and final remarks.

4.1 APPLYING INSTANCE POINTCUTS FOR PROGRAM COMPREHENSION

Complex software systems are difficult to understand because they consist of many elements with complicated dependencies [BH13]. A fac-

tor further complicating the comprehensibility is the dynamicity of execution semantics: In object-oriented programming languages, the dynamic type of the receiver object determines which implementation of the called method will be executed. It is generally not possible to determine the dynamic type of an expression just by looking at the source code. For this reason, often dynamic tools are used to observe the program execution for comprehending a program. As such tools are mainly used during debugging, they are often called *debuggers*; we will also use this term throughout this section to refer to tools which in general allow observing and possibly interacting with program executions at runtime.

In object-oriented programming, on the source code level the main abstractions are classes (object-oriented languages which are not class-based provide an equivalent like prototypes in delegation-based languages). At runtime, however, the main building blocks are *objects* which are instances of classes. Comprehending the behaviour of a program means to comprehend the interplay of objects. In general, there can be arbitrarily many instances of each class, which makes the comprehension difficult as their connections and dependencies can be manifold.

A natural technique to increase comprehensibility is introducing limited number of categories, grouping all objects into categories and considering only the categories during the comprehension task. This is even well supported by object-oriented languages, since classes already form such categories and there is an easily observable relation between an object and its class, and vice versa. However, the categorisation offered by the class structure of the program is not always the best fit for the comprehension task at hand. On one hand, instances of the same class may be used in different ways throughout the program execution; on the other hand, instances of different classes may be used in the same or a similar way.

For these reasons, we claim that another, more powerful and flexible way for dynamically categorising objects is needed to guide the task of program comprehension. We have found that the abstraction provided

by instance pointcuts can be useful to support comprehension of highly dynamic object-oriented programs.

Using pointcut-based techniques for specifying breakpoints in a debugger has been proposed before. For example, Chern and De Volder [CDV07] have proposed to define breakpoints based on the current control flow, similar to AspectJ's `cflow` pointcut designator. Bodden has proposed stateful breakpoint [Bod11], which allows programmers to specify the order in which different events must occur to lead to suspension of the program execution. Yin et al. have presented a language for specifying breakpoints [YBA13] which is also based on a pointcut language and already allows specifying breakpoints based on object relations. Nevertheless, the instance pointcuts approach is the first to actually maintain a set of objects, which relate by how they have been used in the past, and to expose these sets to different debugging facilities.

In the following, we present a walkthrough of a few particular comprehension tasks (Section 4.1.1), we illustrate the applicability of instance pointcuts. Throughout the following section we sketch a user interface that visualises instance pointcuts. In section 4.1.2 we discuss challenges in realising the outlined concepts.

4.1.1 *Example Walkthrough*

In this section, we describe an existing system and in the following sections we discuss several comprehension scenarios where instance pointcuts can be useful. Intertwined with these scenarios, we discuss a hypothetical extension to the Eclipse debugger for using instance pointcuts.

We made the observation that instance pointcuts can support program comprehension while working on an extension to the Jikes Research Virtual Machine [AAB+05]. In particular, the optimising just-in-time (JIT) compiler of this virtual machine was being extended. The optimising compiler works in multiple phases, whereby the first one creates an object-based intermediate representation (IR) from the Java byte-code of the method which is currently compiled. This IR, basically a linked list

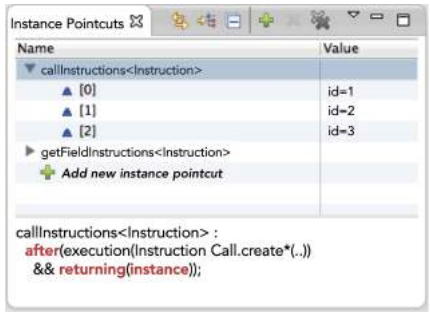


Figure 4.1: The Instance Pointcuts View

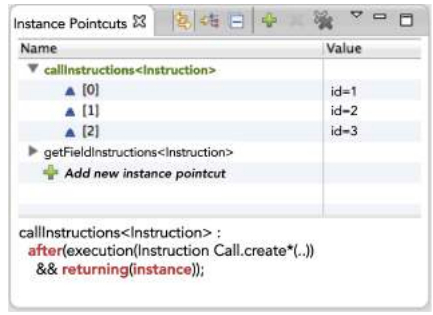


Figure 4.2: A highlighted instance pointcut.

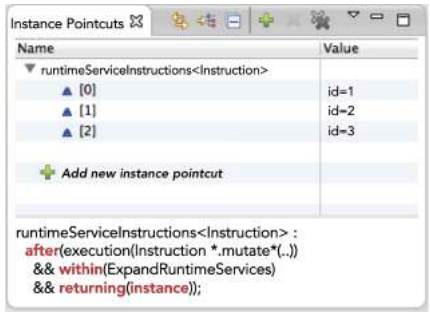


Figure 4.3: Another example of an instance pointcut definition.

of `Instruction` objects, is iteratively analysed and rewritten until eventually machine code is emitted. In that extension we manipulated this IR by inserting instructions and adding rewrites. When we made a mistake runtime failures occurred and we had to understand the impact of this extension on the further processing of the JIT compiler, i.e. to identify the fault we had to comprehend a very complex system.

All elements in the IR are instances of the same class (`Instruction`) which are configured with an `Operator` object determining the behavior of the instruction. Thus, at first sight it is not clear which instruction is represented by an `Instruction` object. Besides the referenced `Operator`

object, also the creation site of the `Instruction` object allows to draw conclusions about the purpose of an `Instruction`: There is a factory class for each kind of instruction. For instance, if we are only interested in instructions representing a function invocation, we can focus our comprehension task on those `Instruction` objects that are returned by a `create` method in the class `Call`.

4.1.1.1 *Scenario 1*

A VIEW FOR INSTANCE POINTCUTS An instance pointcut, let's call it `callInstructions`, selecting the objects described above can be defined as seen in Figure 4.1[1]. The figure depicts a view for defining instance pointcuts. The new view allows to add instance pointcuts which are maintained during the execution of the program, and to inspect the content of the defined instance pointcuts while the program execution is suspended at a breakpoint. The left column shows the names of the defined instance pointcuts. A row with an instance pointcut can be expanded, for example in the figure the instance pointcut `callInstructions` is expanded and its elements are listed. When a row containing the name of an instance pointcut is selected, the detail pane (at the bottom) shows its definition. When a row is selected which represents an element of an instance pointcut, the `toString()` of the value is shown.

LINKING THE INSTANCE POINTCUT VIEW The left-most icon of the toolbar in Figure 4.2 allows linking the Instance Pointcut View with other debugging-related views. When linking is turned on and an object is selected on another view, e.g. in the Variables view, all instance pointcuts are highlighted which contain the selected object. Figure 4.2 shows the instance pointcut `callInstructions` in green, which means that the user has currently selected an object contained in `callInstructions`. In the outlined comprehension task, this feature makes it simple to identify for

1 For brevity we only show simple class names.

each `Instruction` object, if it represents an instruction relevant for the task, i.e., for which an instance pointcut was defined.

4.1.1.2 *Scenario 2*

Besides representing different (virtual) machine instructions, `Instructions` also need to be distinguished by the purpose for which they have been created. The majority of the instructions are directly compiled from the Java byte-code instructions. But some instructions have to be inserted into the generated machine code to inject the runtime services offered by the virtual machine, e.g., thread switching, memory management, and profiling for facilitating optimisations.

A simple example is insertion of the runtime service for managed memory allocation. This is done in a compiler phase after the initial creation of the `Instruction`-based intermediate representation. When an instruction representing the allocation of an object is encountered in this phase, the `Instruction` is *mutated* by turning it into a Call instruction invoking the virtual machine's function realising this service. Figure 4.3 shows the instance pointcut selecting all `Instructions` that inject a runtime service.

USING INSTANCE POINTCUTS IN CONDITIONS AND EXPRESSIONS

Besides using instance pointcuts during inspection when the execution is suspended at a breakpoint, defined instance pointcuts can also be used for defining conditional breakpoints. As an example, consider the task of comprehending a late compiler phase. At this time, many `Instructions` have been mutated for optimization purposes or for injecting runtime services. But many `Instructions` still simply reflect the functionality directly specified by the Java byte-code instructions. Assume we want to comprehend the impact of injected runtime services on the liveness analysis, a phase necessary for generating meta-data needed by the garbage collector. Figure 4.4 shows a screenshot of the breakpoint properties dialog where the condition refers to the instance pointcut defined in Figure 4.3. The execution is only suspended at this breakpoint when

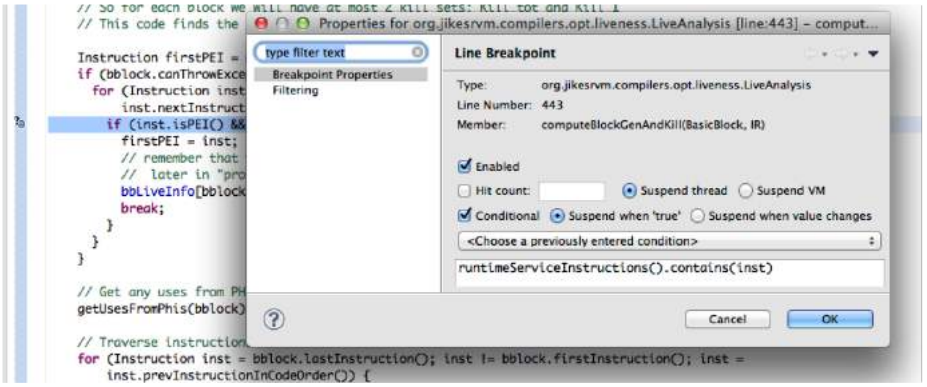


Figure 4.4: Breakpoint properties using an instance pointcut.

the local variable `inst` is selected by the instance pointcut describing the `Instructions` that realize runtime services. In the same way, watch expressions entered in Eclipse's Expression View can refer to instance pointcuts, simply treating them as `java.util.Sets`.

4.1.1.3 *Scenario 3*

INSTANCE POINTCUT WATCHPOINTS To demonstrate the applicability of *remove expression* feature of instance pointcuts, consider the two scenarios above. Since mutation changes the operation represented by an `Instruction` object, mutated `Instructions` must be removed from the instance pointcut representing a specific operation as mentioned in the first scenario. Figure 4.5 shows the extended instance pointcut definition.

Furthermore, the language concept of instance pointcuts allows to advise the joint points when a new instance is added to an instance pointcut and also when an instance is removed. Instance pointcuts internally maintain multisets; we specify that only the initial addition and the final removal of an object are join points, i.e. when the cardinality of an object in the multiset changes from 0 to 1 and from 1 to 0, respectively. These join points can also be used to define watchpoints. The two right-most columns in Figure 4.5 represent this feature. In the example, the watch-

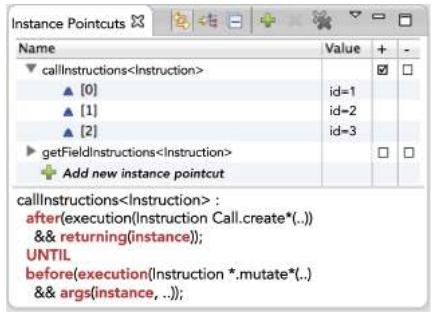


Figure 4.5: Setting watchpoints for instance breakpoint changes.

point for adding an object to the instance pointcut `callInstructions` is enabled; the watchpoint for removing an object is disabled.

4.1.2 *Challenges*

Obviously, the expressiveness of the pointcut language determines how precisely sets of objects can be selected. Since in our prototype we use the AspectJ pointcut language as a base, we are limited to selecting the supported events and specifying supported restrictions. In the example of the optimising compiler of the Jikes virtual machine, this can be a significant limitation: long switch statements identify different ways of processing instructions and it is often relevant in the context of which *case* an object was used. However, since AspectJ pointcuts cannot refer to switch cases, they cannot be used in the add or remove expressions of instance pointcuts.

We are used to adding breakpoints or watch expressions dynamically, during the runtime of the debugged program. With instance pointcuts, this is not possible or at least risky. As is always the case with dynamic deployment of aspects, it may be that some relevant join points have already passed at the time of deployment. Thus, it may be that objects, which should have been selected by an instance pointcut, are not selected; or not all of the recursive additions have been tracked and an ob-

ject is removed from the multiset too early. For these reasons, we do not suggest to support dynamic addition of instance pointcuts, even though this may require to restart the program during one comprehension task, if new relevant categorisations for objects are discovered during runtime.

4.2 EVALUATION

In this section we present an evaluation of our approach in terms of improvements on code quality, performance characteristics and discuss the analyses enabled by instance pointcuts. In the code quality discussion (Section 4.2.1) we present a case study performed on the github android application. In Section 4.2.2 we present the performance characteristics of instance pointcuts based not the execution times of add and remove operations and compare our measurements with equivalent Java code. We conclude the evaluation section by discussing what kind of analyses are made possible with instance pointcuts (Section 4.2.3).

4.2.1 *Code Quality*

GITHUB ANDROID APPLICATION STUDY The Git Repository Hosting site called github offers a mobile application to access its services via Android smart phones. We have performed a preliminary study on this application to show the benefits of using instance pointcuts in a real-life application.

An Android application consists of `Activity` objects which are application components users can interact with to perform a task. Each activity is further divided into `Fragments`, which represents a portion of that `Activity`. `Fragments` have a number of states which mark the beginning and the end of their lifecycle. The users of github can be associated with certain organisations, which gives them access to repositories which are not publicly available but only available through organisation permissions.

The classes which were affected by the addition of this feature can be seen on commit with ID 044262d on the git repository of the project [2]. This particular commit includes changes to `HomeActivity`, `RepositoryListFragment`, `MembersFragment` and `UserNewsFragment`. It also adds two new interfaces `OrganizationSelectionListener` and `OrganizationSelectionProvider`. All of these classes except `RepositoryListFragment` belong to the same package.

All of the listed fragments start their lifecycle with a call to the `onActivityCreated` and they end their life-cycle with a call to the `onDetach`. Fragments register themselves as a `OrganizationSelectionListener` when they are created to the `HomeActivity`. In the implementation of these methods we see the same bookkeeping code repeated, this is shown in Listing 4.1. The purpose of this code is to listen to the changes of the selected organisation for the active user and update the related UI fragments accordingly.

```
1  [RepositoryListFragment,MembersFragment,UserNewsFragment] extends
       Fragment implements OrganizationSelectionListener{
2  ...
3    @Override //Fragment
4    public void onDetach() {
5       OrganizationSelectionProvider selectionProvider =
            (OrganizationSelectionProvider) getActivity();
6       if (selectionProvider != null)
7          selectionProvider.removeListener(this);

9       super.onDetach();
10   }

12   @Override //Fragment
13   public void onActivityCreated(Bundle savedInstanceState) {
14      org = ((OrganizationSelectionProvider)
            getActivity()).addListener(this);
15   \\rest of the implementation is different in each class
16   ...
```

2 https://github.com/github/android/commit/044262d

```
17    }

19    @Override //OrganizationSelectionListener
20    public void onOrganizationSelected(final User organization) {...}
21  }
```

Listing 4.1: The piece of code that is repeated throughout the fragment classes

The `HomeActivity` class implements the `OrganizationSelectionProvider` interface which offers the `addListener(OrganizationSelectionListener)` and `removeListener(OrganizationSelectionListener)` methods. The added listeners are kept in the `orgSelectionListeners` set. When a user selects an organisation, each listener is notified in a for loop by calling their `onOrganizationSelected` method.

From this implementation we can see that the bookkeeping of these listeners is scattered among 3 classes, a total of 6 methods. In order to modularise this concern, we have replaced this listener add/remove implementation with an instance pointcut which is shown in Listing 4.2.

```
1  public aspect OrganizationSelectionProviderAspect{
2   static instance pointcut
        orgSelectionListeners<OrganizationSelectionListener>:
3   before(call(* RepositoryListFragement.onActivityCreated(..))
4    call(* MembersFragement.onActivityCreated(..))
5    call(* UserNewsFragement.onActivityCreated(..)) && this(instance))
6   UNTIL
7   before(call(* RepositoryListFragement.onDetach())
8    call(* MembersFragement.onDetach()))
9    call(* UserNewsFragement.onDetach()) && this(instance));
10 }
```

Listing 4.2: Organisation listener bookkeeping with instance pointcuts

This has resulted in the following code quality improvements in the affected classes.

- The overridden `onDetach` method is removed from all fragment classes, since its only purpose was to remove the listeners from

the provider. This is now managed by the `orgSelectionListener` instance pointcut. This is a quality improvement since we were able to eliminate redundant code from these classes. This has reduced a total of 21 lines of code from three classes.

- The first line of the `onActivityCreated` method was removed, reducing 3 lines in total.
- We have removed the `OrganizationSelectionProvider` interface, the implementation of its methods and the `orgSelectionListeners` set from `HomeActivity` class, reducing in total 11 lines of code.
- The removal of the `OrganizationSelectionProvider` interface also makes the dependency to this class obsolete and makes it possible to remove this dependency from the affected `Fragment` classes. The removal of this interface also simplifies the type-hierarchy.

We were able to localise the bookkeeping of the listener in a single module resulted in more maintainable code; instead of altering three different classes the developer can write a single instance pointcut. Besides modularising the implementation of this concern, the solution with instance pointcuts is also more concise. The 35 lines of the scattered implementation were replaced by 16 lines for an aspect with an instance pointcut. This reduced the size of the implementation of this concern by more than 50

4.2.2 *Performance Evaluation*

We have conducted a performance experiment for measuring the add and remove operations of instance pointcuts. We also measured the performance of equivalent plain Java code and compared both in terms of execution time. We have created four micro-benchmarks for measuring the execution time for each operation with two different cases. These cases are:

ADDING/REMOVING UNIQUE OBJECTS Adding unique objects to the instance pointcut set will result in a growing data structure with objects of cardinality value one, similarly removing these objects will shrink the data structure.

ADDING/REMOVING THE SAME OBJECT Adding the same object more than once to the instance pointcut set will result in a set with a single element and a cardinality greater than one. The remove operation will only reduce this cardinality, until the cardinality becomes zero. Then the object is removed from the set.

We have used the caliper [Goo] micro-benchmarking framework for measuring the execution times. Caliper provides three measuring instruments; allocation, runtime and arbitrary. Since we were interested in the execution times, we used the runtime instrumentation. We have used the following runtime instrumentation options:

- Warmup: This option indicates the minimum time that should elapse before any measurements are taken. We have used the default value of 10 seconds as a warmup period.
- Timing Interval: Instead of number of repetitions, caliper takes a timing interval as an input. Using this value caliper calculates the number of repetitions for an experiment. Higher values mean more precision since the benchmark is less vulnerable to fixed costs. The default value is 500 ms, however we have chosen 1000ms.
- No. of Measurements: Caliper records the final "N" measurements, we have configured caliper to record 10 measurements

In addition to these options we have enabled the option for running the garbage collector before each measurement. Note that this does not guarantee that the garbage collector will run, however we can decrease the chances of its running during measurements.

We have performed these experiments on a laptop running MacOSX v10.9.1 with 2.4 Ghz Intel Core i7 processor and 8GB of RAM. We used

Eclipse IDE (Kepler Service Release 1) for running our benchmarks on JRE 1.7.0u40 64-bit.

Benchmarks are parametrised to be run with $n \in \{1, 10, 100, 1000, 10000\}$ operations and runs for both instance pointcut and Java implementations. Therefore each benchmark yields 10 measurements, 5 for the instance pointcut and 5 for the Java implementation. We show these measurements as generated by caliper, the runtime value corresponds to the time elapsed for n operations and the median of the last ten measurements.

ADDING THE SAME OBJECT The measurements for adding the same object to the instance pointcut set is shown in Figure 4.6. These results show that in general Java and instance pointcuts perform very similarly, with instance pointcuts performing a bit worse occasionally.

SCENARIO.BENCHMARKSPEC.PARAMETERS.SIZE	SCENARIO.BENCHMARKSPEC.METHODNAME	RUNTIME (NS)
1	testAddSameIpc	35.864
1	testAddSameJava	36.827
10	testAddSameIpc	369.966
10	testAddSameJava	368.300
100	testAddSameIpc	3,642.220
100	testAddSameJava	3,597.256
1000	testAddSameIpc	36,343.086
1000	testAddSameJava	36,719.268
10000	testAddSameIpc	361,327.507
10000	testAddSameJava	356,892.757

Figure 4.6: Benchmark results for adding the same object

ADDING UNIQUE OBJECTS In Figure 4.7 benchmark results are shown for adding unique objects. These results also do not show a big performance difference between Java and instance pointcuts.

REMOVE THE SAME OBJECT Caliper provides a set-up method which can be used to prepare the values before each experiment. In this bench-

SCENARIO.BENCHMARKSPEC.PARAMETERS.SIZE	SCENARIO.BENCHMARKSPEC.METHODNAME	RUNTIME (NS)
1	testAddUniqueIpc	35.458
1	testAddUniqueJava	35.746
10	testAddUniqueIpc	361.996
10	testAddUniqueJava	358.938
100	testAddUniqueIpc	3,777.122
100	testAddUniqueJava	3,793.334
1000	testAddUniqueIpc	38,376.621
1000	testAddUniqueJava	37,949.409
10000	testAddUniqueIpc	358,660.568
10000	testAddUniqueJava	363,596.690

Figure 4.7: Benchmark results for adding unique objects

mark we have used this method to populate the data structures, in order to perform and measure the remove operation. Figure 4.8 shows the results for this benchmark. In this case Java performs slightly better than instance pointcuts, however we do not observe a substantial difference.

SCENARIO.BENCHMARKSPEC.PARAMETERS.SIZE	SCENARIO.BENCHMARKSPEC.METHODNAME	RUNTIME (NS)
1	testRemoveSameIpc	3.606
1	testRemoveSameJava	3.234
10	testRemoveSameIpc	22.710
10	testRemoveSameJava	21.250
100	testRemoveSameIpc	217.895
100	testRemoveSameJava	214.691
1000	testRemoveSameIpc	2,099.218
1000	testRemoveSameJava	2,147.226
10000	testRemoveSameIpc	33,514.881
10000	testRemoveSameJava	32,199.211

Figure 4.8: Benchmark results for removing the same object

REMOVE UNIQUE OBJECTS According to the measurements of this benchmark (Figure 4.9), we observe similar results as in removing the same object benchmark.

SCENARIO.BENCHMARKSPEC.PARAMETERS.SIZE	SCENARIO.BENCHMARKSPEC.METHODNAME	RUNTIME (NS)
1	testRemoveUniqueIpc	3.561
1	testRemoveUniqueJava	3.106
10	testRemoveUniqueIpc	34.684
10	testRemoveUniqueJava	32.512
100	testRemoveUniqueIpc	223.287
100	testRemoveUniqueJava	218.276
1000	testRemoveUniqueIpc	2,083.182
1000	testRemoveUniqueJava	2,073.725
10000	testRemoveUniqueIpc	33,313.249
10000	testRemoveUniqueJava	32,815.726

Figure 4.9: Benchmark results for removing unique objects

From these results we conclude that the instance pointcut implementation of add and remove operations, which are the basic operations for the book keeping concern, do not introduce a performance overhead. However these measurements are not sufficient for deducing the overall performance of instance pointcuts. In order to have a better understanding on instance pointcut performance, we must also perform measurements for the following features; type refinement, expression refinement and composition. Due to time limitations these measurements are currently not available; in future work we will address the performance testing in more detail.

4.2.3 *Enabled Analyses*

A declarative syntax such as that of instance pointcuts, generally allows not to perform various checks, to generate well-placed error or warning markers and informative error/warning messages. In the following we discuss checks we deem useful and possible to implement based on our language.

NON-EXISTENT/INCOMPATIBLE TYPES If the type declared by the instance pointcut does not exist, this is a compilation error. Instance

pointcuts provide an additional check during type refinement; it is a compilation error if the refinement type is not a subtype of the referenced pointcut's declared type. In the Java solution the error marker is placed at the line of the `instanceof` check, without giving any context to why the `instanceof` check is performed. The instance pointcut error marker is placed at the line of the refinement, with an elaborate error message explaining the type mistake.

Type compatibility is also an issue while composing instance pointcuts. If in a composite instance pointcut declaration, the instance type is explicit, then the type compatibility between the composite and the component instance pointcuts' is checked. The compiler infers (see Section 3.4.4.3) the appropriate type for every composite instance pointcut, whether the instance type is explicit or not. The computed type is then compared to the declared type; the declared type must be the same or a super type of the computed type. If there is an incompatibility, we put an error marker at the line where the type was declared, indicating the composed type is not compatible with the declared type of the composite instance pointcut.

EMPTY SETS The add expression of an instance pointcut is responsible for populating the instance pointcut set. If the pointcut expressions defined in the add expressions do not match any join-points then it is guaranteed that the instance pointcut set will always be empty. This case is displayed as a compile-time warning which indicates that no objects will be selected. During the expression refinement, composition of the new pointcuts and the referenced one may result in non-matching pointcut expressions.

DOUBLE SELECTION The instance pointcut syntax allows to select the same join-point in both before and after events in the same subexpression. In Listing 4.3 such a case is illustrated. The `Product` object is added before it is applied a discount, and once again after the discount is applied. This may have two different side effects depending on the

definition of the `hashCode` method for the selected type. Either it may add the object to the instance pointcut set as separate instances or it may increment the cardinality of the same object twice. The analogous case exists for the remove expressions. This behaviour is consistent with an instance pointcut's regular behaviour, but we still raise a separate warning for the add and remove cases to notify the developer.

```
1 static instance pointcut surpriseDiscountDouble<Product>:
2   before(call(* ProductManager.submitDiscount(..))
3   && args(instance, SurpriseDiscount))
4   ||
5   after(call(* ProductManager.submitDiscount(..))
6   && args(instance, SurpriseDiscount))
```

Listing 4.3: Adding the same object before and after the same join-point

EXPRESSION REFINEMENT CHECKS During expression refinement there is a special case when the refined instance pointcut is referencing a non-existent event selector and the || boolean operator is used. Assume that `ipc1` only has an `add_before` expression. While refining this instance pointcut the developer mistakenly writes the following:

```
1 static instance pointcut ipc2<T>: add_before:ipc1.add_before
2   add_after: ipc1.add_after || call(..);
```

This definition refers to the non-existent `add_after` expression. This is a compile error since, while the `call` pointcut is selecting joinpoints, no objects are bound. This is illegal to have in an instance pointcut expression. Note that if the operation was && there would be no compile error since the `add_after` expression of `ipc2` would simply be empty and `ip2`'s add expression would only comprise of `ipc1`'s `add_before` event selector.

4.3 RELATED WORK

AO-extensions for improving aspect-object relationships are proposed in several studies. Sakurai et al. [SMU+06] proposed Association As-

pects. This is an extension to aspect instantiation mechanisms in AspectJ to declaratively associate an aspect to a tuple of objects. In Association Aspects the type of object tuples are declared with a `perobjects` clause and the specific objects are selected by pointcuts. This work offers a method for defining relationships between objects. Similar to association aspects, Relationship Aspects [PN06] also offer a declarative mechanism to define relationships between objects, which are crosscutting to the OO-implementation. Relationship Aspects focus on managing relationships between associated objects. Bodden et al. [BSH08] claim that the two above approaches lack generality and propose a tracematch-based approach. Although the semantics of the approaches are very similar, Bodden et al. combine features of thread safety, memory safety, per-association state and binding of primitive values or values of non-weavable classes. Our approach, also extending AO, differs from these approaches since our aim is not defining new relationships but using the existing structures as a base to group objects together for behavior extensions. Our approach also offers additional features of composition and refinement.

The "dflow" pointcut [KM04] is an extension to AspectJ that can be used to reason about the values bound by pointcut expressions. Thereby it can be specified that a pointcut only matches at a join point when the origin of the specified value from the context of this join point did or did not appear in the context of another, previous join point (also specified in terms of a pointcut expression). This construct is limited to restricting the applicability of pointcut expressions rather than reifying all objects that match certain criteria, as our approach does.

Another related field is Object Query Languages (OQL) which are used to query objects in an object-oriented program (e.g., [Clu98]). However OQLs do not support event based querying, which selects objects based on the events they participate in, as presented in our approach. It is interesting to combine instance pointcuts with OQL. For example instance pointcuts can be used as a predicate in OQL expressions, in order to select from phase-specific object sets.

Type States [DF04] allow to define a state chart for a type. This specifies which states an object of that type can be in and what causes state transitions. Similar to our approach, state transitions are triggered by runtime events. However, unlike instance pointcuts, in type states an objects state can only change at method calls where the object is the receiver; with instance pointcuts, objects life-cycle phases can be defined more flexibly by referring to any event where the object is in the dynamic context, e.g., passed as argument or result value. The purpose of the type states approach is to facilitate more powerful invariant checking at compile-time, whereas we provide a mechanism to actually track object sets at runtime. It would be interesting to investigate possibilities for combining both approaches in the future.

4.4 CONCLUSION

In this part we have presented instance pointcuts, a specialised pointcut mechanism for reifying phase-specific object sets. Our approach provides a declarative syntax for defining events when an object starts or ends to belong to a life-cycle phase. Instance pointcuts maintain multi-sets providing a count for objects which enter the same life-cycle phase more than once. Instance pointcut sets can be accessed easily and any changes to these sets can also be monitored with the help of automatically created set monitoring pointcuts. The sets can be declaratively composed, which allows reuse of existing instance pointcuts and consistency among corresponding multi-sets. Finally, we have presented our modular compilation approach for instance pointcuts based on AspectJ and the ALIA4J Language implementation architecture. ALIA4J provided us with the flexibility AspectJ lacked in instance pointcut composition and type refinement.

We have given two examples for the application of instance pointcuts. The first one is the online shop example, where we have illustrated how integrating unanticipated concerns into legacy code can lead to scattered and tangled code. We have especially looked at the case of bookkeeping

certain objects and how such bookkeeping cannot be modularly handled by OOP or traditional AOP. We have also shown that solutions in these paradigms are not reusable.

The second example is the application of instance pointcuts in the program comprehension domain by showing an extension to the Eclipse debugger. Our vision is that by using instance pointcuts during debugging, we will be able to identify objects based on how they are used which is not observable by the class structure. Inspired by the challenges encountered while debugging an extension of the Jikes VM's optimising compiler, we devised three scenarios (Section 4.1.1). In the first scenario we explained how the instance pointcuts concept is useful during debugging; we can easily identify objects that are relevant to the debugging task by looking if they are contained by a certain instance pointcut. In the second scenario we explored the use of instance pointcuts with conditional breakpoints; instance pointcuts can be referenced as `Sets` and we can observe if an object is added to the instance pointcut set by means of a conditional expression. In the third and the final scenario we have utilised the removal feature of instance pointcuts; here instance pointcuts are used to maintain a set of objects starting from the time they were `created` until they were `mutated`. We also showed how the built-in joinpoints for add and remove operations can be used to define watchpoints. Our experience showed that despite some limitations, instance pointcuts are beneficial for program comprehension since they provide the means to create meaningful runtime categories.

The syntax and expressiveness of instance pointcuts partially depend on the underlying AO language; this is evident especially in our usage of the AspectJ pointcut language in the specification of events. Since AspectJ's join points are "regions in time" rather than events, we had to add the "before" and "after" keywords to our add and remove expressions. Thus, compiling to a different target language with native support for events (e.g., EScala [GSM+11] or Composition Filters [BA01b], the point-in-time join point model [MEY06]) would influence the notation of these expressions.

We think the instance pointcut concept is versatile and can be useful in various applications. Instance pointcuts provide another dimension of modularisation which can also be used in conjunction with design patterns. It eliminates boilerplate code to a great extent and provides a readable syntax. We believe that the reuse and composition mechanisms offered by instance pointcuts are beneficial for software evolution since they make it easy to create tailored variations according to new requirements.

Part III

zamk: AN ADAPTER-AWARE DEPENDENCY INJECTION FRAMEWORK

In the evolution of complex software usage of third party components is common practice. While businesses save time and money by reusing existing components, there is still a substantial development effort going into the integration of such components into a given system. In this part we present our contribution,the *zamk* framework, which improves the integration process of externally developed components. For this framework we have developed an external dependency injection language called *Gluer*. This language is used in conjunction with *converters*. Converters are reinterpreted adapters, which are used to convert one type to the other. *Gluer* is a converter-aware language, meaning it can convert injected objects into other types before the injection, given the target injection field is of an incompatible type. Using *zamk* framework developers are able to separate the concerns of the component integration into adaptation and glueing.

5

ZAMK FRAMEWORK

5.1 INTRODUCTION

Software components are self-contained entities with a well-defined interface and behaviour. For two components to co-operate they must be *integrated* through a compatible interface [Weg96]. The integration task also entails creating dependencies between the components, i.e. assigning objects to fields in order to communicate data. The assignment operation requires that the object and the assigned field are of a compatible *type*. When components are developed separately, compatibility may not hold. Therefore it should be, often manually, established through additional programming. This is the second challenge identified in Chapter 2.

The problem of incompatible interfaces is a common software engineering problem and a possible solution is captured in the *adapter pattern* [GHJV95, CMP06]. The adapter pattern describes how to make two types compatible with each other. When an object makes a reference to a data value it requires a specific type, which we refer to as the *target type*. Another object which represents the desired data and behaviour but is an instance of the so-called *source type*, different from the target type, is made compatible with an *adapter*. An adapter introduces a level of indirection between the types whose interfaces are incompatible with each other. An adapter implements the methods of the target type by invoking methods defined in the source type. There are two types of adapters; class adapters and object adapters. Class adapters use multiple inheri-

tance to subclass both the source and the target types. Object adapters wrap the source object and subclass the target type; they implement the target methods by invoking the methods of the source object.

There are some limitations of the traditional adapter pattern. A class adapter is easily implemented with a language that supports multiple inheritance. Developers using single inheritance programming languages need to use workarounds to achieve the same effect. For example in Java multiple inheritance is simulated with the usage of interfaces since a class can implement multiple of them. In order to inherit from two concrete types using Java, we need to extract the interface of one of the types and add this interface to the list of super-interfaces. However, we do not always have control over source code which would allow these operations. The implementation of the object adapter pattern is more flexible, since it only needs to inherit from the target type. In both cases the adapter class is an additional type that is introduced to the source code.

The integration code contains a direct reference to the adapter type instead of just the source and the target types, which introduces additional maintenance efforts. Another problem is related to the *reuse of adapters*. The developer may not possess the knowledge of which adapter is suitable for the adaptation task at hand. This is a problem when adapters are not planned to be reusable and documented poorly. Then one has to search the type hierarchy to find the classes implementing the target type; among these classes the suitable adapter has to be found. The developer may even have to identify if the class she found is indeed an adapter class, if it is not clear from the naming. This is an ad-hoc process that is error-prone as well as time consuming.

So far we have established that integrating two separately developed components requires making their interfaces compatible and assigning object values to the appropriate field thereby building the necessary dependencies. The latter is implemented in the *glue code*. In component-based design loose-coupling is an important principle. Dependency Injection (DI) [Fow04] is a lightweight method for keeping modules loosely

coupled by delegating the creation of concrete objects to so-called *injectors*. This approach allows creating loosely coupled dependencies. When using the adapter pattern, the injection specification is coupled to the adapter implementation. This means during development time the adaptation and the binding tasks are not independent of each other; the developer who is writing the glue code should be aware of the adapters that are available to her.

Let us summarise the problems we have identified above:

P-1 Implementation of the adapter pattern can be hindered by programming language properties.

P-2 The adapter pattern introduces additional dependencies, which increases maintenance efforts.

P-3 Reusing adapters requires additional knowledge and effort from the side of the developer to be used properly.

P-4 The glue code contains dependencies to the adapter classes, which hinder loose coupling.

In light of these issues, we have devised the requirements below, which refer to each of the problems listed above, respectively.

R-1 It must be possible to implement adapters without inheritance. (P-1)

R-2 The adaptations must be performed without creating dependencies to specific adapter classes. (P-2)

R-3 Adapters must be found automatically given a source object and a target type. (P-3)

R-4 Glue code must not contain references to concrete adapters; it must be possible to implement these separately. (P-4)

In order to satisfy these requirements we have designed the *zamk* framework, which unites DI with *under-the-hood* adaptation logic. In our

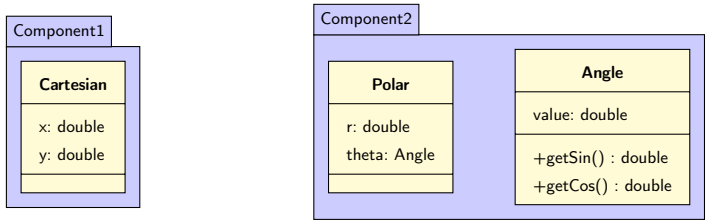


Figure 5.1: UML diagram for the two components

framework we do not employ the traditional adapter pattern but we have created the concept of *converters*, which are user-defined classes that do not have to inherit from a target type to realise the adaptation. *zamk* comes with its own DI mechanism that is used with a designated domain-specific language called *Gluer*. The DI logic is intertwined with the adaptation logic which uses the conversion registry to perform automated adaptation between source and target types. We automate the adaptation process by exploiting the type hierarchies and provide checks and context-relevant messages for correct integration. The details of the framework are explained throughout the chapter.

5.2 MOTIVATING EXAMPLE

In the previous section we have identified the problems attached to the traditional adapter pattern and we have come up with requirements for a solution that would remedy these problems. This chapter focuses on the object adapter pattern, since Java does not support class adapter pattern very well. However the identified problems apply to both of the adapter patterns.

In this section we illustrate the problems mentioned above with an example. Assume we have a plot drawing software which uses the Cartesian coordinate system to represent the points in the plot. The software includes a data component which contains a class called `Cartesian` which has two fields `x, y`, that represents the values on the x-axis and y-axis

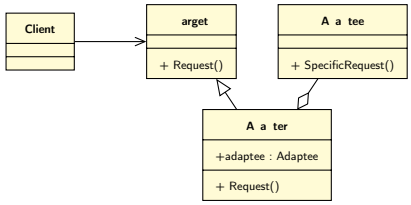
respectively. This class also includes getters and setters for these fields (`Component1` in Figure 5.1). A new requirement is received which states that the software must also support polar coordinates, and the user must be able to view plot points in a selected view (`Cartesian` or `Polar`).

In order to support polar coordinates, a new component which contains classes to represent such data is introduced (`Component2` in Figure 5.1). The class `Polar` contains two fields, `r` representing the radius and `theta` of type `Angle`. This component is to be integrated with `Component 1`. Consider we want to integrate both components using the traditional adapter pattern. It should be possible to obtain the `Polar` representation of any `Cartesian` object by using an *adapter*. This solution is discussed in the following.

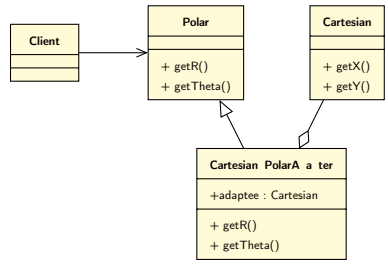
An abstract view of the traditional object adapter pattern is shown in Figure 5.2a. The adapter pattern relies on inheritance and adds a level of indirection between the `Client` and the `Target`. The application of this pattern to the example case requires creating a Cartesian to Polar adapter (`Cartesian2PolarAdapter`) which takes a `Cartesian` object as an *adaptee* and extends the `Polar` class to override its methods (Figure 5.2b). An implementation for this adapter is given in Listing 5.2c.

A conventional implementation of the adapter pattern requires boilerplate code. An improvement over this implementation is proposed by Hannemann and Kiczales [HK02]. In this study they propose a reusable implementation to the class adapter pattern using inter-type declarations. According to the auxiliary code they provide with this study, they propose the adaptee class should subclass the target class, which results as the diagram shown in Figure 5.3a. For our example case the `Cartesian` class will directly have to subclass the `Polar` class and implement its method using its `x, y` field values. This is depicted in the aspect shown in Figure 5.3c. The inter-type declaration on line 3 declares the inheritance relation and the subsequent method implementations are woven into the `Cartesian` class.

The obvious problem with this implementation is that, since `Polar` is a concrete type, it would quickly become unusable due to Java's single



(a) Abstract view of the object adapter pattern



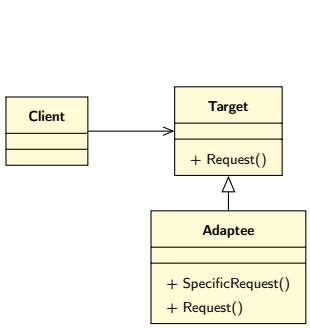
(b) Concrete view of the object adapter for the example case

```
1  public class Cartesian2PolarAdapter extends Polar{
2    Cartesian adaptee;
3    public Cartesian2PolarAdapter(Cartesian c) {
4      this.adaptee = c;
5    }
6    public double getR()
7    {
8      return Math.sqrt(Math.pow(adaptee.getX(), 2) +
           Math.pow(adaptee.getY(), 2)));
9    }
10   public Angle getTheta()
11   {
12     return new Angle(Math.atan(adaptee.getY()/adaptee.getX())));
13   }
14 }
```

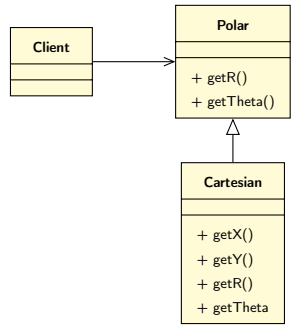
Listing 5.1: Implementation

(c) The implementation for the `Cartesian2PolarAdapter`

Figure 5.2: The diagram of the object adapter and the corresponding Java implementation



(a) Abstract view of the object adapter pattern with inter-type declarations



(b) Concrete view of the aspect-oriented adapter pattern for the example case

```
1  public aspect Cartesian2PolarAdapter {

3    declare parents: Cartesian extends Polar;

5    public double Cartesian.getR()
6    {
7      return Math.sqrt(Math.pow(this.getX(), 2) +
           Math.pow(this.getY(), 2));
8    }

10   public Angle Cartesian.getTheta()
11   {
12     return new Angle(Math.atan(this.getY()/this.getX()));
13   }
14 }
```

Listing 5.2: Implementation

(c) The implementation for the `Cartesian2PolarAdapter` in AspectJ

Figure 5.3: The diagram of the object adapter and the corresponding Java implementation

inheritance. Another limitation comes from the semantics of inter-type declarations. Only *sibling* types, i.e. types which share a parent class, can be used in an inter-type declaration. In this example both classes' super-type is `Object`, which allows us to use the inter-type declaration.

Up to now, we have explained the issues related to the implementation of adaptation concern, we also need to bind the newly added component to our application. In order to bind the polar coordinates into our plotting software, we need to alter some code. According to the selection made in the GUI, the information box should display the coordinate value in the correct format. An example for the integration code is given in Listing 5.3. On line 7 we see an explicit reference to the class `Cartesian2PolarAdapter`; this adapter is introduced at the beginning of the section in Figure 5.2c. Referring to a low-level implementation type hinders software evolution because we need to change code when another adaptation should be used. The other issue is that in order to reuse existing adapters, the developer has to have knowledge about these adapters, or she needs to search to see if a suitable one exists. The explicit reference on line 7 is also a disadvantage for the binding process since how the `Polar` field is initialized is fixed with this reference.

```
1  public void viewPointValue(Point selected){
2
3    if(GUI.format == CARTESIAN)
4      GUI.createNewValueBox(selected.loc(),
           selected.getCoordinates().toString());
5    else if(GUI.format == POLAR)
6    {
7      Polar p = new Cartesian2PolarAdapter(selected.getCoordinates());
8      GUI.createNewValueBox(selected.loc(), p.toString());
9    }
10 }
```

Listing 5.3: The integration of Polar coordinates

In this section we have used the plotter example to illustrate the problems we have mentioned in Section 5.1. Firstly we have described how programming languages can affect the implementation of the adapter

pattern by using Java and AspectJ (P-1). Secondly we have shown the disadvantages of introducing adapter dependencies in the integration code, by means of a new extension to the plotter software (P-2). Thirdly, using the same example, we have discussed why the developer needs adapter specific knowledge and how it slows down the integration process (P-3). Lastly, we have discussed the disadvantage of having an explicit dependency in the glue code (P-4).

5.3 THE *zamk* FRAMEWORK

In this section we explain the details of the *zamk* framework which is designed according to the requirements specified in Section 5.1.

zamk is a development framework specifically tailored for reusable adapters and adapter-aware DI. It offers a new and a lightweight way of defining adapters; these lightweight structures are called *converters*. Converters contain the logic that is used to convert one type to another. Typically this logic defines how an object of a target type can be initialised using the values provided by an object of a source type (e.g., calculating the polar coordinate values from Cartesian values). Converters are provided by the developer and they do not need to subclass any of the types to implement adaptations. Converter classes are stateless. The *zamk* runtime is responsible for finding the correct converter, given a source and a target type. An adaptation requires an intermediate object which references the adaptee and extends the target type; a conversion is merely a method call, to which the adaptee is given as a parameter and which returns an object of the target type. This means the only additional dependency we have to include in the implementation is the *zamk* runtime API. Since the user does not have to refer to specific adapters, *zamk* allows separation of adaptation and binding concerns during integration. *zamk* also comes with its own binding language, called *Gluer*, which uses DI under-the-hood. *Gluer* is a domain-specific language and its declarative nature allows compile-time checks. The user is flexible in

how she chooses to use *zamk*, she can either use the *Gluer* language or she can call the runtime API directly.

Figure 5.4 shows an overview of the steps and the user-defined input that are necessary to perform adaptation using *zamk*.

DEVELOPMENT TIME The goal of the *development time* is to provide the necessary input to *zamk*. In Figure 5.4 we show four inputs that are fed to the *zamk* compile-time; they are (in the order shown in Figure 5.4):

Gluer files: Gluer files contain the *Gluer* statements that define which objects are going to be injected to which fields. *Gluer* statements are used to generate conversion requests, which are *zamk* API calls that trigger the *zamk* adaptation workflow. The *Gluer* language is discussed in Section 5.3.2.1.

Application Classes: These classes are scanned to retrieve information for integration. The retrieved information is used in checking. Application classes are also instrumented with conversion requests that are derived from *Gluer* statements.

Converters: Converters contain methods necessary to convert one type to the other. They are annotated in order to be registered by *zamk* runtime. These are discussed in Section 5.3.2.2.

Converter Precedence Declarations: These are used to resolve conflicts when more than one conversion is found for a specific request. In this case the conversion which is contained by the converter class that has the higher precedence is used.

It is also possible to write plain Java code that calls the *zamk* API directly. However these statements are not processed during compile-time. That is why we do not list them as compile-time inputs. During the *final compilation* phase, they are compiled with all the code that is provided by the user and generated by *zamk*.

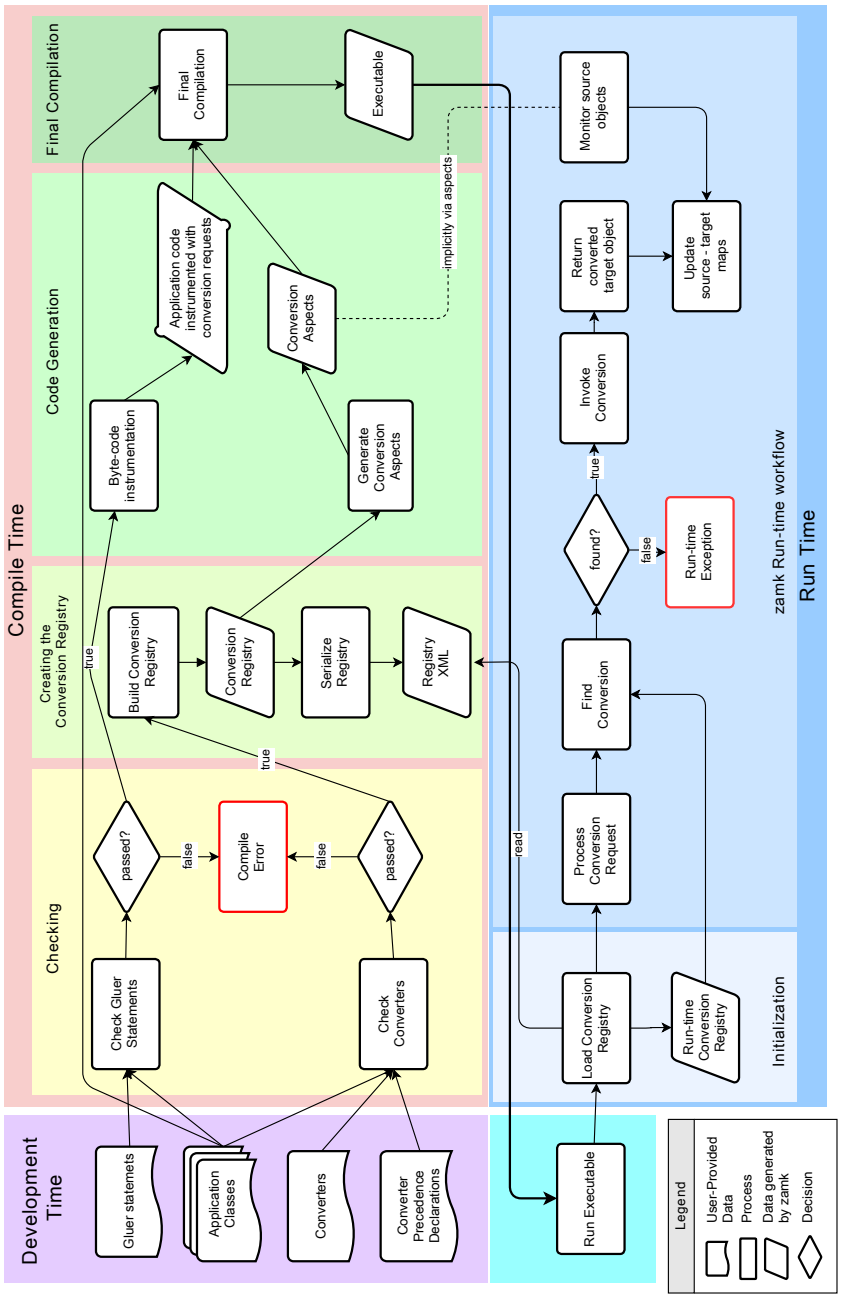


Figure 5.4: An overview of the *zamk* framework

COMPILE-TIME The goal of the compile-time stage is to check the provided input and generate the necessary code using the information derived from the input. Once the input is provided, the *zamk* compile-time workflow starts. This workflow is composed of the following sequential phases as in upper-half of the Figure 5.4:

Checking: There are two checkers; one responsible for checking the *Gluer* files and the other for converter classes. The gluer checker performs the syntax checking; it also checks if the references made in the *Gluer* statements actually exist in the application code. The converter checker performs type checking and well-formedness checking. If any of these checks fails, a compile-error is produced.

Conversion registry: When the converter checking is finished without any problems, *zamk* builds a conversion registry[1] in the form of a data structure to be used in the next step of compile-time workflow: *code generation*. During this phase the conversion registry data structure is also serialised in XML format creating the registry.

Code Generation: This step consists of two separate generation processes. Byte-code instrumentation is responsible for inserting *zamk* conversion requests to the places indicated as the binding points defined in the *Gluer* statements, i.e implementing the DI. The conversion registry that is created in the previous step is used to generate the *conversion aspects*. The conversion aspects are responsible for monitoring the source objects which are associated with a target object.

Final Compilation: When the compile-time workflow is complete a final compilation step is performed, which makes sure the instrumented application classes and the generated files do not contain any errors. At the end of this step we obtain *zamk* runtime ready code, in Figure 5.4 we assume the final compilation product is executable.

1 One converter class may include multiple conversions

The *zamk* compile-time produces two outputs; registry which will be loaded during *runtime initialisation* and the compiled code of *zamk* generated classes and application classes.

RUNTIME During runtime *zamk* processes conversion requests. The *zamk* runtime consists of two parts, a one-time *initialisation* and a *runtime workflow* which is executed for every conversion request (lower-half of Figure 5.4).

Initialisation: The *zamk* runtime starts with an initialisation step, which loads the conversion registry that is serialised during compile-time. The loading process produces a data structure called runtime conversion registry, which is used by the runtime workflow to locate conversions.

Runtime workflow: The compiled program contains *zamk* conversion requests which trigger the conversion finder. A conversion request contains the source object and the desired target type to which the source object should be converted. The *find conversion* process searches for the correct conversion by using the type information included in the conversion request. If a suitable conversion cannot be found, *zamk* produces a runtime exception indicating the error. When a conversion is found, one of two things can happen: 1. If the requested conversion was never performed before for the given source object and the target type, the suitable conversion method is invoked and an object of target type is created. This pair of objects, i.e. the provided source object and the associated target object, is referred to as a source–target pair. 2. Alternatively, *zamk* may find that a request to the same target type was processed before with the given source object. In that case the existing target object is retrieved and returned.

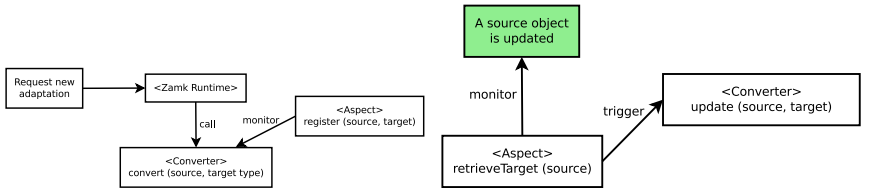
5.3.1 *Using Conversions Instead of Adapters*

In the traditional adapter pattern, the adapter class refers to an adaptee object; if this adaptee's value is changed, it directly affects the return values of the methods that are implemented by the adapter. In order to eliminate the dependency to the adapter type, we have created the notion of conversions. Conversions are defined in user-provided converter classes. Conversions simulate adaptation as a one-time conversion and a series of updates during an object's life-cycle.

A conversion consists of two parts. The first part consists of two user-defined methods: a convert and an update method. The second part is an *aspect*, called conversion aspect, generated by *zamk* which contains a hash-map of source (adaptee) and target objects. The conversion aspect is generated during compile-time[2]. The responsibility of this aspect is to monitor the source (adaptee) objects and update the target objects if a source object changes. The update operation is performed by calling the user-defined update method. It is also possible to have two-way adaptations, in which case the conversion aspect is responsible for keeping track of conversion in the opposite direction.

In Figure 5.5a the first step of the conversion process is shown. A new conversion is requested by giving a source object and a target type. In the traditional adapter pattern this source object is the adaptee and an adapter which is a subtype of the target type is instantiated that aggregates this source object. In our conversion process this source object is passed onto the convert method of an appropriate conversion (automatically found by the framework, see Section 5.3.3.3) and this method returns the corresponding target object, which is initialised according to the values provided by the source object. When a new target object is created, the *zamk* runtime registers the source-target pair in a map. This map represents the *has-a* relationship between the adapter and the adaptee.

2 Refer to the code generation step of Figure 5.4



(a) A new adaptation request (b) Keeping source and target objects synchronized

Once a target object is linked to a source object, they are kept synchronised. This is ensured by the generated conversion aspect. The conversion aspect monitors the events which change the source objects, and when such an event is encountered it retrieves the corresponding target object. Then it triggers the *update* method in the converter class to update the target object that is linked to the changed source object.

In this method of adaptation, there is no need to create an intermediate adapter type. The converter directly creates an object of the target type and keeps it up-to-date. As we explain in Section 5.3.2, the user only has to define a conversion in a converter class.

A limitation of converters is the case where no public constructors or fields are available for the target type. A traditional object adapter does not have this limitation, since it extends the target type and has access to protected members as well as the public ones. In order to workaround this limitation it is possible to declare an adapter as a converter. An example implementation can be seen in Listing 5.4. Note that this implementation may still run into limitations in the update method caused by private fields, i.e. hidden object state.

```
1 @Converter
2 public class Cartesian2PolarAdapter extends Polar{
3   Cartesian adaptee;
4   public Cartesian2PolarAdapter(Cartesian c) {
5     this.adaptee = c;
6   }
7   public double getR(){..}
```

```
8   public Angle getTheta(){..}
9   @Convert
10  public static Polar cart2polar(Cartesian c){
11    return new Cartesian2PolarAdapter(c);
12  }
13  public static void updatePolar(Polar registryObject, Polar
        newValue){..}
14  }
```

Listing 5.4: An object adapter defined as a converter for converting a Cartesian object to a Polar object

5.3.2 *Compile-time*

In this section we explain the elements and modules that are involved during the compile-time of the framework. In Figure 5.5 a simplified version of the compile-time workflow is shown. In this figure we have numbered the steps taken during compile-time, since they are performed sequentially.

As mentioned before, the developer is responsible for providing converters that are specific to her application. The converters are required to adhere to a specific structure, which is discussed in Section 5.3.2.2. The correctness of the converters are checked by the process Check Converters, in the Checking step. All of the processes in this step take the application classes as input, since the checking operation uses these classes to investigate the existence of the dependencies. *Gluer* files are checked by the Check Gluer Statements process. The checking of the gluer files and the inputs/outputs are shown in dashed style, since providing *Gluer* files is an optional step. As mentioned before it is also possible to call *zamk* API directly from Java.

Once the provided input is checked for errors, the checked converter classes are input to the Build Conversion Registry process, inside the Creating the Conversion Registry step. The processes included in this step are also numbered, since the Serialize Registry process requires the con-

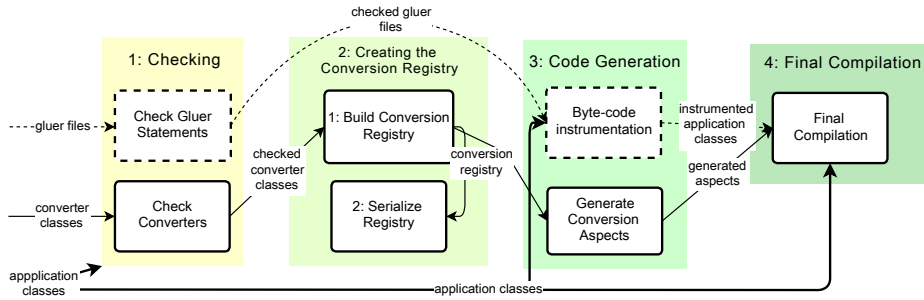


Figure 5.5: The compile-time workflow and dataflow of *zamk*

version registry data structure produced by the Build Conversion Registry process.

The conversion registry data structure is also passed to the third step Code Generation, which contains two processes. First one is the optional process Byte-Code Instrumentation, which takes the checked *Gluer* files as input. This step is not performed if no *Gluer* files are provided. The second one is the Generate Conversion Aspects process, which takes the conversion registry as input and outputs the conversion aspects.

The fourth and the final step is the compilation of all code that is generated by *zamk* and provided by the developer.

5.3.2.1 *Gluer DSL*

The *Gluer* language is a concise DSL that is designed to declare dependencies between fields and objects. We have developed the *Gluer* language to offer an external, non-intrusive way of declaring bindings. Essentially *Gluer* is an external DI declaration language which creates the objects to be injected and is connected to an adaptation logic which can process the created objects before the injection happens.

A parser for *Gluer* (we use the *gluer* extension to denote Gluer specifications) is implemented in Clojure [Hico8], a Lisp dialect compiled for the JVM [RHB13]. One of the design goals was to have the DSL grammar to be easily extendible at compile-time, and even at runtime. This way,

support for new keywords and their behaviour can be added by loading plugins at runtime, making the framework extendible.

We make use of Clojure's dynamic dispatch features to facilitate this extensibility with respect to the types of *target* and *source* selection statements [3].

SYNTAX AND SEMANTICS We chose the keyword `glue` instead of `inject` since *Gluer* does more than DI.

The syntax of the *Gluer* statements is defined as follows:

```
glue <target-field> with
[[new|single] <source-class> | retval <Java-expr>]
[using <converter>]
```

<TARGET-FIELD> The target field is a fully qualified name of a non-static field of a class. It can refer to any object type.

<SOURCE-CLASS> The source class represents the fully-qualified name of the class to be instantiated and injected to the target field. There are several options for creating this objects.

NEW The `new` statement is followed by a fully classified name of a class. This means, whenever an object is to be injected, it should be newly created using the *default constructor* of the source class.

SINGLE Similar to `new`, **`single`** statement is followed by a fully qualified name of a class, which is instantiated when an injection is triggered. The difference is instead of creating a new object each time, a *single* object is reused among injections.

RETVAL Short for "return value", this keyword is followed by a Java expression, which returns the object we would like to inject. When

3 The source code of the *Gluer* parser is available on GitHub: https://github.com/aroemers/gluer. Note that this version corresponds to the language explained in our PPPJ'13 paper

an injection is triggered, the method is called and the returned value is glued to the injection field.

USING The `using` keyword is optional and can be used to override the automated converter finding logic. With this keyword the user can point to a specific converter to be used while converting the source object to the target type, before the injection.

CHECKS *zamk* performs some compile-time checks to ensure the correctness of the *Gluer* statements. The checks which result in compile errors contain specific information about the place and the cause of the error.

- Target-field is checked to see if it actually exists.
- For creating the source object with the `new` and `single` keywords, the framework requires that the source class contains a public no-argument constructor. For the `retval` keyword, the framework checks if the referred method exists.
- The `using` keyword triggers two checks. The first one checks if the referred converter exists and the second one checks if any of the conversions in that converter is suitable for converting from the source class to the target field.
- If the *Gluer* statements are error free up to this point, then a conflict check is performed to see if any two *Gluer* statements try to inject into the same field.

5.3.2.2 *User-defined Converters*

The users of *zamk* are responsible for creating converters.

A converter satisfies three important requirements:

1. It must be annotated with the `@Converter` annotation

2. It has to include at least one conversion which is composed of two *static* methods:
    a) A *convert* method that is annotated with `@Convert`. This method takes a single parameter and must return an object value.
    b) An *update* method that is annotated with `@Update`. This method takes two parameters of the same type and does not return any value.
3. There cannot be two convert methods that have the same argument and return types in a single converter. The same is true for the update method.

Since the methods are annotated there are no restrictions imposed by *zamk* on the method naming. The convert method contains the logic for converting a source object to a target object. It takes a source object as its single argument and creates the corresponding target object. This method is invoked by *zamk* when an adaptation is requested for a *new* source object, i.e a source object that has not been adapted before to a given target type. If the requested adaptations have been performed before and *zamk* has a matching source–target pair in the registry, then the existing target object is returned.

The update method contains the logic for updating a target object. It takes two arguments of the type target; the first is the existing registry object which will be updated due to its associated source object's state change. The second argument is the updated state of the registry object based on the new state of the associated source object. In the update method, the developer specifies how an object of the target type is assigned a new state. The reason we update the values of an existing object is, if we simply replace the object with a new one then the dependencies to the old object will be outdated. Since the references to this object would still point to the old version, whereas the new version is stored in a new address location, unknown to the dependents of the old one.

It is also possible to declare two-way conversions in a converter class. For a conversion from A-to-B, if the inverse conversion B-to-A is defined in the same converter class, then *zamk* registers this as a two-way conversion.

Referring back to our example given in Section 5.2, a user-defined adapter for the `Cartesian-Polar` conversion that conforms to the requirements above can be defined as shown in Listing 5.5. In this example we have defined two methods; `cart2polar` which is annotated as the convert method. It takes a source object of type `Cartesian` and returns a `Polar` object, the `Polar` object is created using the source object (lines 6 –8). The second method is `updatePolar`, which updates the `registryValue Polar` object using the field values of the `newValue Polar` object. If we add the convert and update methods for `Polar` to `Cartesian` conversion to the converter in Listing 5.5 then *zamk* will register a two way conversion between these types. From the developer's perspective implementation requirements do not change, *zamk* handles the operations required to keep converted objects synchronized.

```
1  @Converter
2  public class Cartesian2PolarUser{
3   @Convert
4   public static Polar cart2polar(Cartesian source)
5   {
6     double r = Math.sqrt(Math.pow(source.getX(), 2) +
          Math.pow(source.getY(), 2))
7     Angle a = new Angle(Math.atan(source.getY()/source.getX()));
8     return new Polar(r,a);
9   }
10  @Update
11  public static void updatePolar(Polar registryObject, Polar
        newValue)
12  {
13    registryObject.r = newValue.r;
14    registryObject.the = newValue.the;
15  }
```

```
16 }
```

Listing 5.5: A converter defined for converting a Cartesian object to a Polar object

This converter can additionally contain include methods to convert from different types. The `@Converter` annotation simply marks a class to be found by *zamk*. When *zamk* finds a converter class, it expects that it has one or multiple *pairs* of convert and update methods. If a convert method is found to be without an update method or vice versa, this results in a compilation error. The convert-update method pairs must be declared in the same converter. *zamk* does not merge methods from separate converter classes.

When a pair of convert and update method is found, type checks are performed. In order to register a conversion *zamk* looks at the convert method's source (single parameter) and target (return) types. The accompanying update method *must* take arguments of exactly the target type, since *zamk* is only able to pair an update method to a convert method by looking at the argument types. Otherwise a compilation error indicating the situation is given to the user.

5.3.2.3 *Conversion Registry*

During compile-time a converter registry is created and serialised using the annotated converter classes.

This procedure assumes the converter class is structurally correct, i.e. the methods are properly annotated and there is exactly one matching update method for each convert method in a converter. The createRegistry procedure first finds all classes annotated with `@Converter`. From each converter class it finds the convert and update methods and puts them into separate lists. For each convert method found in the converter the variables `targetType` (the return type of the convert method) and `sourceType` (the argument type of the convert method) are initialised. Then the matching update method is found by iterating over the list of update methods and comparing their argument with the target types.

Procedure 1 Creating the conversion registry

```
1:  procedure createRegistry
2:      converterClasses ← all classes annotated with @Converter
3:      for all class ∈ converterClasses do
4:          convertMethods ← @Convert methods in class
5:          updateMethods ← @Update methods in class
6:          registryPerClass ← empty set
7:          for all convertMethod ∈ convertMethods do
8:              sourceType ← the argument type of convertMethod
9:              targetType ← the return type of the converMethod
10:             for all updateMethod ∈ updateMethods do
11:                 if updateMethod.argumentType = targetType then
12:                     conversion ← (sourceType, targetType,
    convertMethod, updateMethod)
13:                     add conversion to registryPerClass
14:                 end if
15:             end for
16:         end for
17:         registerTwoWayConversions(registryPerClass)
18:         add (c.FQN, registryPerClass) to registry
19:     end for
20:     serialise(registry)
21: end procedure
```

When an update method is found the record of the conversion is created and registered.

The procedure for creating the converter registry can be seen in Procedure 1. In this procedure the outermost for loop is iterating over all of the classes that are annotated with `@Converter`. The two inner for loops are for finding the convert - update pair per converter class. Once these are found a *conversion* is created and added to *registryPerClass* set.

The *registryPerClass* data structure contains the list of *conversion* items per converter class. A *conversion* entry consists of the source and the target types, the name of the convert and update methods. The update method included in the created *conversion* are removed from the corresponding lists. When all conversions in a converter is found, the *registryPerClass* for a single converter class is complete. However at this point the *registryPerClass* does not contain any information about two-way converters.

The `registerTwoWayConversions` (shown in Procedure 2) procedure processes the *registryPerClass* list and changes its contents if it contains any two-way conversions. This procedure iterates over the conversion list *registryPerClass* and tries to find conversions which have the inverse source and target types. Once a pair of such conversions are found, a *newConversion* which contains the source and the target types and the convert and update method names of both conversions is created. The *newConversion* items are collected in a separate set called *newRegistryPerClass* and the individual conversions forming a two-way conversion are marked in the *registryPerClass* list. After all two-way conversions are found, the marked entries from *registryPerClass* are removed and the new entries collected in *newRegistryPerClass* are added to list *registryPerClass*. This procedure is also responsible for assigning the unique IDs to each conversion, which is done in the for loop shown on line 16. This unique ID is later used in the aspect generation and is a combination of the converted types and the converter class' simple name which contains this conversion. Once the *registryPerClass* list is in its final form it is mapped to the *registry* with the converter's

fully-qualified name as the key. After the list of converters is exhausted the registry is fully populated. The last operation is the serialisation of the registry to an XML file, which is done by the serialise operation at the end of the Procedure 1.

Procedure 2 Finding the two-way conversions

```
1: procedure registerTwoWayConversions(registryPerClass)
2:     newRegistryPerClass ← empty set
3:     for all unmarked x ∈ registryPerClass do
4:         for all unmarked y ∈ registryPerClass do
5:             if y.sourceType = x.targetType then
6:                 if y.targetType = x.sourceType then
7:                     newConversion ← (sourceType, targetType,
   x.convert, x.update, y.convert, y.update)
8:                     mark x and y
9:                     add newConversion to newRegistryPerClass
10:                end if
11:            end if
12:        end for
13:    end for
14:    remove marked conversions from registryPerClass
15:    newRegistryPerClass ← newRegistryPerClassregistryPerClass
16:    for all conversion ∈ registryPerClass do
17:        assign uid to conversion
18:    end for
19: end procedure
```

The registry is serialised as an XML file, with the format shown in Listing 5.6. The XML structure adheres to the converter class structure; multiple conversion tags are enclosed with a converter tag, which takes the fully-qualified name of the converter as a value. The conversion tag marks if the conversion is a two way conversion and contains the source–target types for the conversion, and the convert–update method

```
<converter = [FQN]>
  <conversion uid=".." twoway = [true|false]>
    <source>[source-type]</source>
    <target>[target-type]</target>
    <convert value = "1">[convert-method]</convert>
    <update value = "1">[update-method]</update>
    <!--For two-way conversions-->
    <convert value = "2">[convert-method 2]</convert>
    <update value = "2">[update-method 2]</update>
  </conversion>
  <conversion...
</converter>
```

Listing 5.6: The XML code for a registry item

pair for each conversion direction (source-to-target and target-to-source). The convert and update tags contain the method names for convert and update methods respectively. The tags marked with *1* are the methods responsible for the conversion from source-to-target, and following tags marked with *2* are the convert and update methods for the inverse conversion, if the conversion is indeed a two way conversion.

5.3.2.4 *Code Generation*

There are two separate code generation modules included in *zamk*. The first one is the byte-code generation and weaving module which is used to generate code from *Gluer* statements. The second one is the aspect generator, which uses user-defined converters to generate conversion aspects for each converter.

BYTE-CODE GENERATION AND INSTRUMENTATION The *Gluer* statements are parsed and transformed into Javassist [CN03, Chi13] library calls, which is used to insert byte-code into class files. Since *Gluer* is a proof of concept implementation it only supports constructor injections;

```
1  public abstract aspect ZamkAbstractAspect<T, U> {
2    pointcut updateObserver(T obj): set(* T.*) && target(obj);
3    pointcut updateObserver2(U obj): set(* U.*) && target(obj);
4    Map<T, U> map;
5    after(T obj): updateObserver(obj)
6    {
7      if (map.containsKey(obj)) {
8        this.update(obj);
9      }
10   }
11   abstract <T> void update(T obj);
12 }
```

Listing 5.7: The abstract reusable aspect `ZamkAbstractAspect`

the target field of the injection is initialized during object creation with this type of injection. It is also possible to extend the grammar and the byte-code generator to implement setter injections.

ASPECT GENERATOR For each conversion in a user-defined converter a specialized aspect is generated. Every generated aspect extends the abstract aspect `ZamkAbstractAspect`. `ZamkAbstractAspect` uses the generics support in AspectJ to define a reusable aspect. This generic aspect is shown in Listing 5.7. The pointcut `updateObserver` selects the join-points, where the fields of an object of generic type `T` are *set*. The pointcut `updateObserver2` does exactly the same thing for the type `U`. This pointcut is used when an aspect is generated for a two-way conversion. This aspect also declares a map, from `T` to `U`; i.e. from source to target type. The after advice between lines 5– 10, checks if the map contains the `obj` and calls the abstract method `update`. The `update` method is generated specific to each aspect.

The template for the generated aspect is shown in Listing 5.8. `[simpleName]` is the user-defined converter's class name. We concatenate the source and the target type names to create `[source-type]2[trgt-type]` and

```
1  public privileged aspect
       [simpleName][source-type]2[trgt-type]GenAspect extends
       ZamkAbstractAspect{
2    private static String aspectUID = [..];
3    public [name][source-type]2[trgt-type]GenAspect() {
4      ZamkRuntime.register(aspectUID, map);
5    }
6    map = new WeakHashMap<[source-type], [trgt-type]>();
7    @Override
8    <[source-type]> void update([source-type] obj){
9      [name].[update-method](map.get(obj),
           [name].[convert-method](obj));
10   }
11 }
```

Listing 5.8: The code generation template for producing an adaptation-specific aspect

`GenAspect` at the end of `[name]`. Since a single converter class can include multiple conversions, the aspect names also include the type information in their names. This naming convention provides a unique fully-qualified name for each generated aspect.

Each aspect has a unique ID called `aspectUID` (line 2) and a constructor which is called to create a *singleton* instance of the aspect (lines 3–5). The `aspectUID` is the conversion unique ID which is assigned during the generation of the conversion registry and is the fully-qualified name of the aspect. Inside this constructor the aspect registers its `map` to the *zamk* runtime with its unique ID; this map is used for storing the source–target pairs. The instantiation of the inherited field `map` is shown on line 6. It is constructed using generics notation; the `[source-type]` is the type of the source object and the `[target-type]` is the type of the target object. These types are determined by looking at the argument type and the return type of the convert method, respectively. The overridden `update` method is generated to call the update method of the conversion (lines 8– 10).

Note that the update method of a conversion takes two arguments of the target type, the old value and the new value. In the override `update` method of the aspect, this is used as follows; the first argument, the old value, is retrieved from the map by using the source object. The new target object value is generated by calling the convert method on the changed source object. Then these values are passed as arguments to the update method of the conversion.

```
1  public privileged Cartesian2PolarUserCartesian2PolarGenAspect
       extends ZamkAbstractAspect{
2    private static aspectUID = "Cartesian2PolarUserCartesian2Polar";
3    Cartesian2PolarUserCartesian2PolarGenAspect()
4    {
5      ZamkRuntime.register(aspectUID, map);
6    }
7    map = new WeakHashMap<Cartesian, Polar>();

9    <Cartesian> void update(Cartesian obj)
10   {
11     Cartesian2PolarUser.updatePolar(map.get(obj),
           Cartesian2PolarUser.cart2polar(obj));
12   }
13 }
```

Listing 5.9: The aspect generated for the Cartesian to Polar converter

The generated aspect for the plotter example's converter shown in Listing 5.5 is shown in Listing 5.9. The `updateObserver` pointcut, declared in the parent aspect, monitors all the `Cartesian` objects and selects the join-points where they are changed. The `update` method, calls the `updatePolar` method of the `Cartesian2PolarUser` converter to update the corresponding `Polar` object (line 9).

In case of two-way conversions the generated aspect slightly changes. Instead of a `WeakHashMap` we use a `HashBiMap` [B+]. A `HashBiMap` has two underlying `HashMaps` with inverse type parameters and it preserves the uniqueness of its values as well as its keys. The constructor of the aspect does not change.

For two way conversions we use both of the pointcuts declared in the parent abstract aspect `ZamkAbstractAspect`. For the second pointcut `updateObserver2` we generate the corresponding advice. Also we generate a second update method for calling the conversion in the opposite direction. The resulting two way aspect for the plotter example is shown in Listing 5.10.

```
1  public privileged Cartesian2PolarUserCartesian2PolarGenAspect
       extends ZamkAbstractAspect{
2   private static aspectUID = "Cartesian2PolarUserCartesian2Polar";
3   Cartesian2PolarUserCartesian2PolarGenAspect()
4   {
5    ZamkRuntime.register(aspectUID, map);
6   }
7   map = HashBiMap.create<Cartesian, Polar>();

9   after(Polar p): updateObserver2(p)
10  {
11   if(map.inverse().containsKey(p){
12    update2(p);
13   }
14  }
15  <Cartesian> void update(Cartesian obj)
16  {
17   Cartesian2PolarUser.updatePolar(map.get(obj),
         Cartesian2PolarUser.cart2polar(obj));
18  }
19  <Polar> void update2(Polar obj)
20  {
21   Cartesian2PolarUser.updateCartesian(map.inverse().get(obj),
         Cartesian2PolarUser.polar2cart(obj));
22  }
23 }
```

Listing 5.10: The aspect generated for the Cartesian to Polar two-way converter

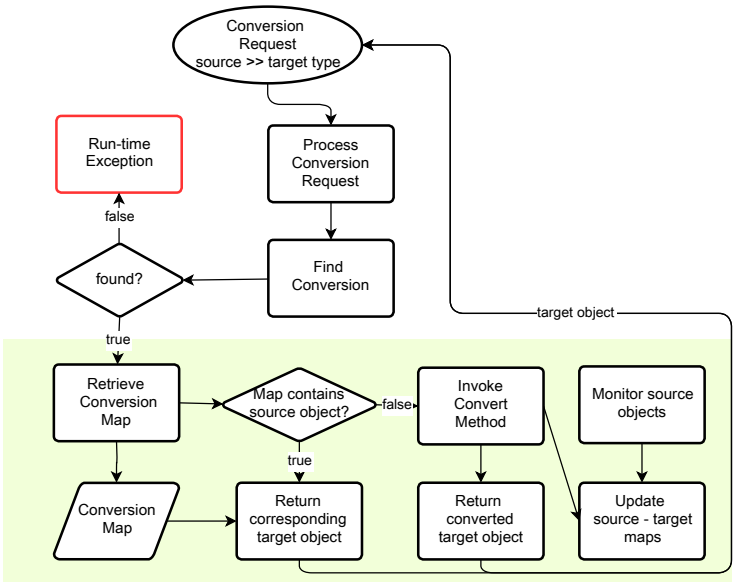


Figure 5.6: The process triggered by a conversion request

5.3.3 *Runtime*

The *zamk* runtime is triggered by conversion requests. There are two ways to create such requests; the first one is the *Gluer* statements which are transformed into *zamk* API calls in the byte-code, the second one is including direct references to the *zamk* API in the base-code. Both of these operations trigger the same conversion finding process.

Figure 5.6 shows how *zamk* processes a conversion request. For each request the appropriate conversion is found by the Find Conversion process. When this process completes successfully, the conversion map of the found conversion is retrieved. If the source object already exists in the conversion map, the corresponding target object is retrieved. Otherwise a new target object is created by invoking the convert method of the conversion. The conversion map is also updated to include the new source–target pair. Once the desired target object is created or retrieved

(in case the given source object is already associated with a target object), it is returned to the owner of the request. The *zamk* runtime is also responsible for managing source–target object pairs, by monitoring the source objects and updating the maps accordingly.

5.3.3.1 *Initialization*

In order to process conversion requests, *zamk* performs a one-time initialisation step at the beginning of the runtime which consists of loading the conversion registry and conversion aspect registration.

zamk creates a registry of conversion during compile-time (Section 5.3.2.3). At the beginning of the runtime this registry is loaded by parsing the XML file which contains the conversion definitions. As mentioned before the XML file contains data about two kinds of conversions, one-way and two-way. Each `<conversion>` tag is mapped to a `Conversion` object, which is the parent type for `OneWayConversion` and `TwoWayConversion`.

From the conversion data two separate `HashMaps` are generated; `conversionMap` and `sourceMap`. The `conversionMap` contains the `Conversions` mapped by their unique IDs. The `sourceMap` is mapping from the source type of the conversion to a list of unique IDs of conversion which convert from the source type declared as the key. The `sourceMap` is constructed to avoid iterating over the whole `conversionMap` while searching for a conversion.

Conversion aspects register their source–target maps during initialization. This is done by calling the `register` method of the `ZamkRuntime` in the constructor of the aspect. The aspects are registered with the unique id and their map. Even though the maps are declared and initialized in the conversion aspects, the contents of the maps are managed by the `ZamkRuntime`.

5.3.3.2 *zamk Conversion Requests*

A conversion request passes on the source object and a desired target type, in return *zamk* runtime provides an object of the target type which

is initialized based on the value of the source object. The conversion requests are communicated to *zamk* by calling the `getConvertedValue(Object, Class<?>)` method, which is a static method of `ZamkRuntime`. The pseudo code for the `getConvertedValue` method is shown in Procedure 3. The first step is to find the suitable conversion for the given input by calling the `findConversion` method (line 3). If this operation is successful, the unique ID of the found conversion is returned and stored in the *uid* variable. The details of the `findConversion` method are discussed in Section 5.3.3.3.

Using *uid* the source–target map for the conversion is retrieved from `mapPerConversion` hash-map (line 4). *zamk* checks if there is a source–target entry in the conversion map for the given source object, i.e. if the source object has been converted before using this conversion. If this is true, then the corresponding target object is retrieved and returned. Otherwise the request triggers a new conversion, then the `invokeConversion` method is called with the *uid* and the source object to create a new target object. The new source–target pair is added to the *map* and the target object is returned.

Procedure 3 The `getConvertedValue` method

```
1:  procedure getConvertedValue(source, targetType)
2:      sourceType ← source.Class
3:      uid ← findConversion(sourceType, targetType)
4:      map ← mapPerConversion.get(uid)
5:      if source ∈ map then
6:          return map.get(source)
7:      else
8:          target ← invokeConversion(uid, source)
9:          add (source, target) to map
10:         return target
11:     end if
12: end procedure
```

5.3.3.3 *Finding a Conversion*

The `findConversion` method used in Procedure 3 implements an algorithm, which uses type information to find the *closest* conversion among *eligible* conversions for the given source type and the expected target type.

Given a conversion request from type X to type Z, the requirements for an an eligible conversion are as follows:

E-1 The conversion source type is exactly the same as or is a super type of type X and,

E-2 the conversion target type is exactly the same as or is a sub type of Z type.

The closest conversion is characterized as follows:

C-1 Among eligible conversions its source-type is the closest to the actual type X object given in the conversion request.

C-2 Among the eligible conversions with the same source-type proximity, its target-type is the closest to the Z type given in the conversion request.

Let us clarify these descriptions with an example. Figure 5.7 shows two separate hierarchies that represent different classifications for animals. Conversions are defined between the types of these hierarchies. Consider the conversion request `getConvertedValue(largeMammal, Vertebrate.class)`. According to our description of the eligible conversions, we can check each conversion to determine if they are in fact eligible. Starting from the bottom of the figure:

- The `L2W` conversion converts from `LargeMammal` to `WarmBlooded`. Since the type of the source object (`LargeMammal`) exactly matched the conversion's source type (`LargeMammal`) it satisfies E-1, since the target type of this conversion is `WarmBlooded` which is a sub-type of `Vertebrate`, the conversion also satisfied E-2. Hence we conclude that the conversion is eligible.

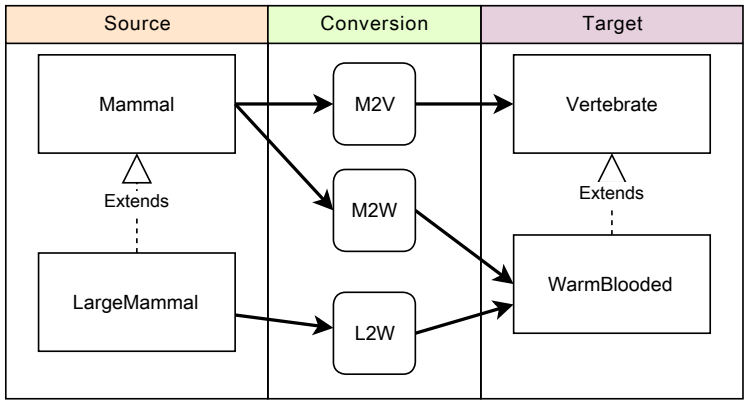


Figure 5.7: Two type hierarchies for representing animals and conversions between them

- The conversion `M2W` satisfies the E-1 since `Mammal` is a super-type of `LargeMammal`, it also satisfies E-2 since its target type `WarmBlooded` is a sub-type of `Vertebrate`.
- Similarly to `M2W` the conversion `M2V` satisfies E-1 since its source type `Mammal` is a super type of `LargeMammal` and its target type `Vertebrate` is exactly the same as the target type given in the conversion request, satisfying E-2.

From this analysis we conclude that all three conversions are eligible for this conversion request.

Conversion	Source	Target
L2W	0	1
M2W	1	1
M2V	1	0

Table 5.1: The type distances of conversion's source–target types to the source–target types given in the conversion request getConvertedValue(mammal, WarmBlooded.class)

To identify the closest conversion, we look at the hierarchical distances of the conversion types to the types given in the conversion request. For this example the distances are listed in Table 5.1. Even though `L2W` and `M2V` have the same combined distance, the closest conversion is `L2W` since closeness of the source type has priority while deciding on the closest conversion. The information of the conversion comes from the source object. For a conversion to be more accurate, the type of the given source object should be as close as possible to the source type it converts from, i.e the source object should be as specialized as possible. That's why the closeness check prefers the closeness of the source type over the closeness of the target type.

These operations are implemented in `findConversion`; pseudo code is shown in Procedure 4. The procedure starts by creating an empty list named *eligibles* which contains the eligible conversions. In order to determine these conversions, we first need to find the super types of the source type by walking their type hierarchy. This is done by the method `getAllSuperTypes` which uses reflection to populate the full set of super types (classes and interfaces) of a given type. On line 3 this method is called and the returned list is stored in *superTypeSource*. Note that when the `java.lang.Object` is passed as an argument to this method, the result contains the single element `java.lang.Object`.

The while loop (lines 4– 16) iterates over the *superTypeSource* set and checks if that source type is associated with any conversions; the `if` statement on line 5 checks if *sourceMap*'s key set contains the *sourceType*. When this expression evaluates to true, *zamk* retrieves the candidate conversions from the *sourceMap* and store them into the list *candidates* (line 6). The *candidates* list contains the IDs of the conversions that are applicable to the *sourceType*. The conversions pointed by *candidates* only satisfy the source type criteria of an eligible conversion, therefore we still need to check the target types of these conversions to detect if they are indeed eligible. This detection is done in the for loop on lines 7– 12. For each unique id in the list *candidates*, we retrieve the corresponding conversion from the *conversionMap* (line 8) and store it into the

Procedure 4 The procedure for finding the most suitable conversion

```
1: procedure findConversion(sourceType, targetType)
2:     eligibles ← empty list
3:     superTypesSource ← getAllSuperTypes(sourceType)
4:     while superTypesSource.hasNext do
5:         if sourceType ∈ sourceMap.keySet then
6:             candidates ← sourceMap.get(source)
7:             for all id ∈ candidates do
8:                 conversion ← conversionMap.get(id)
9:                 if conversion.getTargetType ⊂ targetType then
10:                    eligibles.add(id)
11:                end if
12:            end for
13:        else
14:            sourceType ← superTypesSource.next
15:        end if
16:    end while
17:    if eligibles = ∅ then
18:        Runtime Exception, no conversions found
19:    else
20:        found ← findClosest(eligibles, sourceType, targetType,
   superTypesSource)
21:    end if
22:    if found.size > 1 then
23:        return resolvePrecedence(found)
24:    else
25:        return found(0)
26:    end if
27: end procedure
```

variable *conversion*. Then we check if the target type of this *conversion* is a sub-type of the *targetType* passed to the conversion request (line 9). If this expression is true then we add the id of the *conversion* to the list of *eligibles*.

When the if statement on line 5 evaluates to false, then we set the variable *sourceType* to the next element in *superTypesSource* and reiterate the process until there are no more elements left in *superTypesSource*. At the end of this iteration, we obtain a list of eligible conversions stored in *eligibles*.

The next operation is to find the closest conversion from this list. First we check if the list *eligibles* is empty (line 17); if it is an empty list then we throw a runtime exception, indicating there are no suitable conversions found for the given source and target types. If the list is non-empty then we invoke the method `findClosest` (line 20) and store the returned conversion id values in the variable *found*. It is possible that there are more than one equally close conversions for a given request, if this is the case we invoke to `resolvePrecedence` method (line 23) and return the value obtained from this method. If the `resolvePrecedence` method cannot resolve the ambiguity in the *found* list then it throws a runtime exception, indicating there is not enough information to resolve the precedence. If the `findClosest` method returns a single conversion then we simply return the first element of the list *found*.

`findclosest` METHOD This method finds the closest conversion from the list of eligible conversions. In order to perform this task, the method takes the eligibles list, source and target types and the lists of their super types as an argument. The method creates the table shown in Table 5.1 for each eligible conversion, and decides on the closest according to the closest conversion criteria given at the beginning of this section.

`resolveprecedence` METHOD When `findClosest` method finds multiple equally close conversions for a conversion request, the `resolvePrecedence` method is invoked. The task of this method is to pro-

cess the precedence information given during compile-time and decide which of the conversions should be applied to a conversion request.

5.3.3.4 *Target Object Creation and Retrieval*

In Section 5.3.3.2 we have mentioned the steps taken after a conversion is found. In this section we discuss these steps in detail.

We show a partial pseudo code of the `getConvertedValue` in Procedure 3. When the `findConversion` method returns a conversion id, we use it to retrieve the map for that conversion. Then we check if the map contains the *source* object, if it does we return the target object, associated with this *source*. This operation is called target object retrieval. By having this operation, we ensure that the state of a conversion is preserved, similar to an object adapter inferring its state from its adaptee.

If the *source* object was not converted before with the found conversion, we need to create a new source–target pair. To do this we must call the `invokeConversion` method, which reflectively invokes the convert method of the found conversion. The source code for this method is given in Listing 5.11. The argument `uid` is used to retrieve the conversion from the `conversionMap` (line 2). A `Conversion` object contains the fully-qualified name of the converter it is contained in and the names of the convert and update methods. We first create a `Class` object for the converter that contains the `conversion` (line 3). After constructing an array for argument types of the convert method (line 4), the convert method for the `converter` is loaded from the `Class converter` (line 5). The retrieved method is invoked by calling the `invoke` method and passing on the argument `source` (line 6). The result of the method invocation is returned.

```
1 public static <T, U> T invokeConversion(String uid, U source) {
2   Conversion conversion = conversionMap.get(uid);
3   Class converter = Class.forName(conversion.getConverterName());
4   Class[] argTypes = new Class[] { source.getClass() };
```

```
5   Method convert =
        converter.getDeclaredMethod(conversion.getConvertMethod(),
        argTypes);
6   return (T) convert.invoke(null, source);
7  }
```

Listing 5.11: The `invokeConversion` method which reflectively invokes the convert method of a given conversion

5.3.3.5 *Object Synchronisation*

The source–target pairs kept in the conversion maps are maintained by conversion aspects. We have previously discussed the structure and the members of these aspect in Section 5.3.2.4. A conversion aspect contains a pointcut (`updateObserver`) which monitors all of the changes made to the objects of source type. When this pointcut matches, the changed object is bound. The after advice that uses this pointcut, looks at its source–target map to check if the map contains the bound object as a key. If this is the case, then the corresponding target object is updated by creating a new object of target type and passing the values of the newly created object to the existing target object. The copying of the values is performed by the update method of the conversion.

5.3.3.6 *Runtime API*

zamk runtime API offers two overloaded methods to access its functionality.

- `getConvertedValue(U source, Class<T> targetType)` : This method invokes the fully-automated functionality of *zamk*. The runtime steps mentioned in the sections before take place, and the converted value is returned.
- `getConvertedValue(U source, Class<T> targetType, String using)` : Calling this method is equivalent to adding a `using` clause in *Gluer*. The `String` argument should be the fully-qualified name of the conversion. In this method the `findConversion` is not called.

The runtime API is useful when the developer does not wish to use an external language for DI. The usage of the runtime API makes the dependency explicit but the developer can use it anywhere in the code. The DI mechanism can only create dependencies in the constructor of the object or in the setter methods.

The two provided methods are static methods, so they can be used by referring to the *zamk* runtime as shown in Listing 5.12.

```
1 public void zamkAPIcalls(TypeA typeA)
2 {
3  TypeB typeB = ZamkRuntime.getConvertedValue(typeA, TypeB.class);
4  TypeB typeBUsing = ZamkRuntime.getConvertedValue(typeA,
       TypeB.class, OtherTypeAtoTypeBConverter.class.getName());
5 }
```

Listing 5.12: Using *zamk* API in the implementation

In this example we have used the `getName()` method of the Java `Class` interface to get the qualified name for the desired converter. This implementation will give a compilation error if the class does not exist; this way we can eliminate the problem of type checking that comes with using a `String`. If the referred type given as the using parameter does not contain a method for the conversion, then the developer will see a `RuntimeException` which informs the developer about cause of the error.

6

zamk: DISCUSSION

In this chapter we discuss the applicability of *zamk* to real-life cases by means of a repository analysis. We follow by presenting related work and conclude with our final remarks.

6.1 APPLICABILITY OF *zamk*

We performed repository mining using the *Boa* platform [dye13] to find open source Java projects that have applied the Adapter pattern for (unanticipated) integration of an independently developed component. Boa indexes SourceForge projects and allows querying the revision data in the projects' version control systems. The search was performed by looking at commit messages that hinted at the introduction of the Adapter pattern for integration purposes. The full Boa query can be seen in listing 6.1.

```
1 counts: output collection[string][int] of string;
2 p: Project = input;

4 when (i: some int; match('^java\$',
      lowercase(p.programming_languages[i])))
5  when (j: each int; def(p.code_repositories[j]))
6   when (k: each int;
        match('adapterintegrationpluggablebindingdependency
        injectionwrapperlegacy componentlegacy systemlegacycomponent',
        lowercase(p.code_repositories[j].revisions[k].log)))
```

```
7       counts[p.code_repositories[j].url][p.code_repositories[j]
8           .revisions[k].id] << strreplace(strreplace(p.code_repositories[j]
9           .revisions[k].log, ''\r'', ''\\r'', true), ''\n'', ''\\n'', true);
```

Listing 6.1: The query script for Boa.

The query resulted in 52712 revisions in total, which we have manually inspected further until we have identified two projects that actually have performed an unanticipated integration of the Adapter pattern.

A first example where the Adapter pattern was not used initially is *Argo*[1], 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 the 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 DI.

A downside of the approach taken in the Argo project is the need to update the source code in order to integrate adapters. The current implementation of *Gluer* language only supports injection into fields, while in Argo the dependency is introduced in terms of a method argument. However, as we have discussed before *Gluer* is extensible with new injection points, such that code manipulation can be avoided with an extended *Gluer*. By using *Gluer*, intertwining the core parsing logic with the integration logic can be prevented.

Second, the project *TimeLog Next Generation*[2] 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 Eclipse's `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 6.2 shows how the registration is done programmatically.

1 http://sourceforge.net/projects/argo/
2 http://sourceforge.net/projects/timelogng/

```
public void init() {
      delegate.registerAdapters(new SimpleWorkbenchAdapterFactory(
                  new ClientAdapter()), Client.class);
      delegate.registerAdapters(new SimpleWorkbenchAdapterFactory(
                  new ProjectAdapter()), Project.class);
      delegate.registerAdapters(new SimpleWorkbenchAdapterFactory(
                  new TaskAdapter()), Task.class);
      delegate.registerAdapters(new SimpleWorkbenchAdapterFactory(
                  new DefaultTreeSetAdapter()), TreeSet.class);
      delegate.registerAdapters(new SimpleWorkbenchAdapterFactory(
                  new DefaultListAdapter()), List.class);
      delegate.registerAdapters(new SimpleWorkbenchAdapterFactory(
                  new PeriodAdapter()), Period.class);
}
```

Listing 6.2: TimeLog registering its adapters.

The `init` method in the listing is called when the Eclipse Framework is started. 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.

In this example, registering and using the adapters is done imperatively, as was evident in listing 6.2. This means that if no suitable adapter is available at some point, this is only discovered at runtime. Since *zamk* is also a DI framework and is declarative instead of imperative, it finds such issues before the application is run. This has a huge advantage and shows the strength 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.

6.2 RELATED WORK

The *zamk* framework provides the means to use two component integration methods together; these are adaptation and DI. In the literature

these two approaches have been researched separately; to the best of our knowledge there is not one work that combines the two. However there are studies which are parallel to our work; in this section we will give a background on such similar studies.

In [Gsc12], Gschwind proposes a type-based adaptation framework for adapting component interfaces. An adapter concept called component adapters is introduced, similar to our work a repository of adapters (in our case conversions) is used. This approach uses traditional adapters as the underlying mechanism and provides meta-information about when to use such adapters. The goal of this work is to provide automatic adaptation of components which can be used with component models such as CORBA [Vin97], Enterprise JavaBeans [RB10] etc. This work does not provide a thorough solution to provide automatic adaptation like *zamk* does, however it provides a way to compose adapters together, which *zamk* does not offer.

Mezini et al. discuss so-called *Composite Adapters* [MSUL99], which are a group of concrete adapters that work together. The authors propose to extend OOP languages with constructs that make such composite adapters available to source code that 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. *zamk* also chooses which conversion to use implicitly (unless explicitly given with the `using` keyword). A common goal of of our work and Composite Adapters is to provide non-intrusive techniques for component integration.

In [HA09] Hummel et al. identify parallel challenges to the challenges we have identified. They introduce a new type of adapter called *Managed Adapter* which contains a hash table of adaptees and the adapted target objects. The client uses this class to retrieve adaptees or adapted objects. Although this approach claims to be non-intrusive, still dependencies to new types (i.e. managed adapters) are introduced. Also this approach does not provide an on-the-fly adaptation mechanism as *zamk* does.

Cámara et. al [CCCM07] describe a semi-automatic and non-intrusive approach for supporting commercial off the shelf components' composition and evolution. Similar to *zamk*'s conversion repository, this work makes use of a mapping repository which contains the message mappings between components. The adapter manager uses these mappings to generate adapters [BBC05], which are kept in an adapter repository. In this work the authors use AOP to intercept messages between components. This approach is similar to *zamk* in terms of using a repository of adapters which are managed by aspects. However the approach relies on adapter generation and method interception which adds complexity to the management of the integration.

The Scala language [OSV08] contains a feature called *views*. A view is an implicit conversion which is implemented as a method that takes one type as parameter and returns an object of another type. Such a method is defined with the `implicit` keyword; it is type-checked and implicitly applied by the compiler. This language feature is very similar to what we are trying to achieve by creating conversions and applying them implicitly when a source object is injected to a target field. However implicit conversions in Scala do not maintain the link between the source and the target objects, i.e when the target object is created, its state is not affected by the change in the state of the source object.

The Eclipse Framework also has an adapter repository managed by an *Adapter Manager* [Bea08]. Adapter Manager has methods to register adapter factories with the target and adaptee types. This registration of adapter factories can also be done declaratively in a configuration file. Adapter Manager is also used to retrieve the suitable adapter factory, based on the object and the expected interface. Eclipse's adapter management offers a more dynamic approach than *zamk*, however it lacks the important checking features that *zamk* provides.

There are numerous aspect-oriented dynamic adaptation approaches [YCS+02, Ost02, PS04]. The *zamk* is also related to such approaches since the conversions are performed by aspects. These approaches focus on self-adaptive software where the components are introduced or removed

during runtime. In the *zamk* framework the dependencies are declared at design-time so automatic adaptation is currently not supported. We will discuss a possible extension in Chapter 7 as future work.

We have also used DI [Fow04] in *zamk*. DI has close connections with aspect-orientation. Google Guice [Van08, Goo13] uses AOP to support method interception in order to complement DI. In Guice developers must use the `@Inject` annotation in the source code, therefore changing the code. Chiba and Ishikawa introduce GluonJ [CI05] to provide an AO-language for DI and the implementation of glue code. In *zamk* we have used AOP to manage the source-target pairs, where the target object is the result of a conversion and is injected to a target field.

6.3 CONCLUSION

In this part we have introduced *zamk*, which is an adapter aware DI framework. *zamk* provides an external language for DI, called *Gluer* and it uses under-the-hood adaptation logic for implicitly performing adaptations before an object is injected to a target field. *Gluer* provides a declarative way of defining dependencies between types; since it is an external language no changes in the actual source code are needed for injection. *Gluer* has its own checking mechanism, which informs the user of possible errors such as non-existent targets, methods or conversions.

zamk has its own adaptation mechanism called conversions. Conversions are declared in regular Java classes which are declared to be converter classes with an annotation. The *zamk* runtime is responsible for creating a repository of conversions which are then applied to source objects to obtain objects of the target type. Conversions are declared in converter classes which contain one or more pairs of `convert–update` methods. It is also possible to declare two-way conversions by providing two sets of `convert–update` method pairs. The well-structured definition of converters allows the developer to focus the implementation efforts on what is necessary to perform an adaptation. Converters are concise and light-weight structures that encapsulate the adaptation concern.

zamk keeps track of the source–target object pairs inside aspects, which are generated for maintaining the state relationship between such pairs. This provides a non-intrusive way of maintaining the conversion object's state by monitoring the source object via `set` methods. We have made use of the Java generics support in AspectJ to create a generic aspect which is reused by each generated aspect.

The added benefit of using *zamk* is, while defining dependencies between components, *zamk* can implicitly create the compatible object, by using predefined conversions. In this chapter we have shown that a conversion is a light-weight concept and it does not suffer from the impediments caused by the programming language properties which affect the adapter pattern. We have also shown that a traditional object adapter can also be implemented as a converter, thereby allowing the developer to make use of the adapter pattern. *zamk*'s automatic adaptation mechanism saves the developer from the (often domain-specific) knowledge of which adapter to use. The conversions can be implemented by another developer which has the knowledge to convert from one type to another, and the binding of components can be implemented in *Gluer* by a developer who is knowledgeable in how to integrate certain components. Using the *zamk* the developer only needs to refer to the source and target types; i.e. she does not need to refer to an intermediate adapter type. This feature also reduces the number of type dependencies used in the integration code.

Overall *zamk* provides a flexible and a non-intrusive approach to establish dependencies between components. Using *zamk* framework the development of the integration code is less error-prone and resulting integration code is more maintainable than the traditional approaches discussed in this part.

Part IV

FINAL REMARKS

7

CONCLUSION AND FUTURE WORK

In this chapter we summarise our contributions, discuss the current status of the presented work and elaborate on future directions.

At the beginning of the thesis we have defined two problem statements. In the proceeding chapters we have presented solutions to each of these problems.

- In legacy systems the exposed context is not always sufficient for integrating new functionality. We have identified a need for selecting objects according to their participation in certain events, which are of interest to the new functionality. We have shown a solution for this problem in Chapter 3.
- In order to integrate new software, late in the software life-cycle, we may need adaptations to establish a common interface between the legacy software and the new software. To do this, often a third adapter type is introduced into the programs, increasing the number of dependencies. The approach in Chapter 5 presents a solution for this problem.

7.1 INSTANCE POINTCUTS

In Chapter 3 and Chapter 4 we have introduced a novel AO language feature called instance pointcuts. Instance pointcuts aim to solve the problems related to life-cycle-based bookkeeping of objects. In software, the

life-cycle changes occur when an object participates in an event; however these changes are often implicit. To offer better support for processing objects according to their life-cycle phases, instance pointcuts are used to declare the beginning and the end of a life-cycle as events. Instance pointcuts are reusable constructs, which can be refined by type or events and they can be composed using set operations. The declarative nature of instance pointcuts gives rise to several compile-time checks which are not automatically possible with equivalent imperative code.

In this chapter we have explained the details of the instance pointcuts approach by giving the syntax and semantics for using this language mechanism. We have also introduced a prototype for instance pointcuts as an extension to AspectJ, however the concept itself is applicable to any AO-language.

Instance pointcuts are transformed into general-purpose code by means of code generation. Our goal was to support modular compilation, to this end we have generated code that uses the ALIA4J. Due to the modular architecture ALIA4J provides, we were able achieve this goal.

We have shown the benefits of using instance pointcuts with two examples. We have applied the instance pointcuts approach to a real-life case study the github android application and showed improvement in code quality. In the second example we applied instance pointcuts to program comprehension domain to create meaningful runtime categories. In both of these studies we have shown that instance pointcuts is a versatile concept which is beneficial to use when categorising objects.

Future Work

As future work, our first goal is to use instance pointcuts in a real-life medium size code base and analyse the benefits of this approach better. To do this, first we will determine the bookkeeping code in the software and replace it with instance pointcuts. This refactoring work will shed light on how this new modularisation mechanism improves software. In the next steps of we will analyse several software evolution scenarios to

better illustrate the benefits of the reusability properties instance pointcuts offer.

Our current implementation of the instance pointcuts is based on AspectJ. We would like to implement instance pointcuts as an extension to other AO-languages, for example the event-based language described in [BMAK11]. In this we study we would like to understand how the underlying language can improve the instance pointcut expressiveness.

It is interesting to combine instance pointcuts with Object Query Languages (OQL). OQLs are used to query objects in an object-oriented program (e.g., [Clu98]). OQLs do not support event based querying, which selects objects based on the events they participate in, as in instance pointcuts. For example instance pointcuts can be used as a predicate in OQL expressions, in order to select from phase-specific object sets.

7.2 *zamk*: AN ADAPTER-AWARE DEPENDENCY INJECTION FRAMEWORK

In Chapter 5 and Chapter 6 we have introduced an adapter-aware dependency injection framework called *zamk*. In this work our aim was to maintain loose-coupling while establishing interoperability between software parts. We have identified many problems with the traditional adapter pattern which is usually the solution to interface incompatibility.

We have created the *zamk* framework as an attempt to remedy the problems with software integration at the object level. The *zamk* framework provides an external dependency-injection language called *Gluer* and a runtime which implicitly performs adaptations between the injected object and the target of the injection. *Gluer* is a declarative language which provides a readable format and meaningful keywords for defining dependencies. When declaring a dependency in *Gluer*, one does not have to provide type compatibility between the source object and the target field, since it will be handled by *zamk* later. Therefor *Gluer* provides a more flexible way of declaring dependencies.

We have also created the concept of conversions, which can be used in place of or accompanying traditional adapters. Conversions are light-weight constructs which consist of a convert–update method pair. Convert methods are responsible for converting an abject of the source type to a corresponding object of the target type. The update methods are responsible for maintaining the value of the created target object, should the state of the source object change. The relationships between these source and target objects are maintained by conversion aspects, which are generated during compile-time from conversion definitions. The benefit of conversions is that their simple and well-structured definition allows the developer to focus the implementation efforts on what is necessary to perform an adaptation.

Overall, *zamk* succeeds in providing the developers with an extensive framework for component integration. Using *zamk* framework the development of the integration code is less error-prone and resulting integration code is more maintainable than the traditional approaches.

Future Work

Currently *zamk* does not support chained adaptations; i.e. multiple levels of conversions to obtain an object of the target type. We would like to include this feature in order to make the conversion more reusable. This type of chaining can be type based only and can entail finding the right sequence of converters to call within each other. Precedence information can still be used to select the conversions that will be used in the chain.

Using instance pointcuts with *zamk* we can also further improve the software integration process. In this co-operation, instance pointcuts can provide a set of source objects, which are selected according to their life-cycle phase. Then these specific set of objects can be used for conversions, which will provide a fine-grained mechanism to only adapt and inject relevant objects. This idea is discussed in [HBA12] by us, where we used a different methodology for adaptation.

BIBLIOGRAPHY

[AAB+05] Bowen Alpern, Steve Augart, Steve M. Blackburn, Maria Butrico, Antony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark Mergen, J. Eliot B. Moss, Ton Ngo, and Vivek Sarkar. The Jikes Virtual Machine Research Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44, 2005.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. In *Transactions on Aspect-Oriented Software Development I*, pages 135–173. Springer, 2006.

[AGMO13] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Caesarj website, 2013.

[AWB+94] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In *Object-Based Distributed Programming*, pages 152–184. Springer, 1994.

[B+] K. Bourrillion et al. Guava: Google core libraries for java 1.5+(2010).

[BA99] Barry Boehm and Chris Abts. Cots integration: Plug and pray? *Computer*, 32(1):135–138, 1999.

[BA01a] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[BA01b] Lodewijk M.J. Bergmans and Mehmet Aksit. How to deal with encapsulation in aspect-orientation. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[BBC05] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, 2nd edn. SEI Series in software engineering*. Addison-Wesley Pearson Education, Boston, 2003.

[Bea08] Wayne Beaton. Eclipse corner article: Adapters. https://www.eclipse.org/articles/article.php?file=Article-Adapters/index.html, June 2008.

[BH13] Christoph Bockisch and Kardelen Hatun. Instance pointcuts for program comprehension. In *Proceedings of the 1st workshop on Comprehension of complex systems*, CoCoS '13, pages 7–12, New York, NY, USA, 2013. ACM.

[BM07] Christoph Bockisch and Mira Mezini. A flexible architecture for pointcut-advice language implementations. In *Proceedings of the 1st Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms, VMIL*. ACM, 2007.

[BMAK11] Christoph Bockisch, Somayeh Malakuti, Mehmet Aksit, and Shmuel Katz. Making aspects natural: events and composition. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD*, pages 285–300. ACM, 2011.

[BNGA04] Lodewijk Bergmans, István Nagy, Gürcan Gülesir, and Mehmet Aksit. Compose*: Aspect-oriented composition tools for composition filters [presentation]. 2004.

[Bod11] Eric Bodden. Stateful breakpoints: a practical approach to defining parameterized runtime monitors. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 492–495, New York, NY, USA, 2011. ACM.

[Bru01] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd edition, 2001.

[BSH08] E. Bodden, R. Shaikh, and L. Hendren. Relational aspects as tracematches. In *Proceedings of the 7th international conference on Aspect-oriented software development*, pages 84–95. ACM, 2008.

[BSY+12] Christoph Bockisch, Andreas Sewe, Haihan Yin, Mira Mezini, and Mehmet Aksit. An in-depth look at ALIA4J. *Journal of Object Technology*, 11(1):1–28, 2012.

[CCCM07] Javier Cámara, Carlos Canal, Javier Cubo, and Juan Manuel Murillo. An aspect-oriented adaptation framework for dynamic component evolution. *Electronic Notes in Theoretical Computer Science*, 189(0):21 – 34, 2007. <ce:title>Proceedings of the Third International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT 2006)</ce:title>.

[CDV07] Rick Chern and Kris De Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 96–106, New York, NY, USA, 2007. ACM.

[Chi13] Shigeru Chiba. Javaassist website, 2013.

[CI05] Shigeru Chiba and Rei Ishikawa. Aspect-oriented programming beyond dependency injection. In AndrewP. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume

3586 of *Lecture Notes in Computer Science*, pages 121–143. Springer Berlin Heidelberg, 2005.

[Clu98] S. Cluet. Designing oql: Allowing objects to be queried. *Information systems*, 23(5):279–305, 1998.

[CMP06] C Canal, JM Murillo, and P Poizat. Software Adaptation. *L'objet*, 2006.

[CN03] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 364–376. Springer Berlin Heidelberg, 2003.

[DF04] Robert DeLine and Manuel Fähndrich. Typestates for objects. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer Berlin Heidelberg, 2004.

[Dij82] Edsger W Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[dye13] *Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories*, 05/2013 2013.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, 2004.

[Fra13] Spring Framework. Spring framework website, 2013.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[GLLRK77] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics. v5*, pages 287–326, 1977.

[Goo] Google. Caliper: Microbenchmarking framework for java.

[Goo13] Google. Google guice website, 2013.

[Gsc12] Thomas Gschwind. Automated adaptation of component interfaces with type based adaptation. In Karin K. Breitman and R. Nigel Horspool, editors, *Patterns, Programming and Everything*, pages 45–61. Springer London, 2012.

[GSM+11] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. Escala: modular event-driven object interactions in scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 227–240, New York, NY, USA, March 2011. ACM.

[HA09] Oliver Hummel and Colin Atkinson. The managed adapter pattern: Facilitating glue code generation for component reuse. In *Formal Foundations of Reuse and Domain Engineering*, pages 211–224. Springer, 2009.

[HBA12] Kardelen Hatun, Christoph Bockisch, and Mehmet Aksit. Aspect-oriented language mechanisms for component binding. In *SLE (Doctoral Symposium)*, volume 935, pages 11–14, 2012.

[HdRB+13] Kardelen Hatun, Arjan de Roo, Lodewijk Bergmans, Christoph Bockisch, and Mehmet Akşit. Engineering adaptive embedded software: Managing complexity and evolution. In *Model-Based Design of Adaptive Embedded Systems*, pages 245–281. Springer, 2013.

[Hico8] Rich Hickey. Clojure official website. http://clojure.org, 2008.

[Hiro3] Robert Hirschfeld. Aspects-aspect-oriented programming with squeak. In *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232. Springer, 2003.

[HJSW10] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the gap between modelling and java. In Mark van den Brand, Dragan Gaševic, and Jeff Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 374–383. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-12107-4_25.

[HKo2] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[JHATo9] Rod Johnson, Juergen Hoeller, Alef Arendsen, and R Thomas. *Professional Java Development with the Spring Framework*. Wiley. com, 2009.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. *ECOOP 2001 Object-Oriented Programming*, pages 327–354, 2001.

[KMo4] Kazunori Kawauchi and Hidehiko Masuhara. Dataflow Pointcut for Integrity Concerns. In Bart de Win, Viren Shah, Wouter Joosen, and Ron Bodkin, editors, *AOSDSEC: AOSD Technology for Application-Level Security*, March 2004.

[Kri96] Bent Bruun Kristensen. Object-oriented modeling with roles. In *OOIS?95*, pages 57–71. Springer, 1996.

[Kru92] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.

[Leh96] Manny M Lehman. Laws of software evolution revisited. In *Software process technology*, pages 108–124. Springer, 1996.

[LLO03] Karl Lieberherr, David H Lorenz, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, 2003.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *Software Engineering, IEEE Transactions on*, 33(10):709–724, 2007.

[MEY06] Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. A fine-grained join point model for more reusable aspects. In Naoki Kobayashi, editor, *Programming Languages and Systems*, volume 4279 of *Lecture Notes in Computer Science*, pages 131–147. Springer Berlin / Heidelberg, 2006. 10.1007/11924661_8.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. Endotesting: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2001.

[MSUL99] Mira Mezini, Linda Seiter, Santa Clara Univeristy, and Karl Lieberherr. Component integration with pluggable composite adapters. In *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 1999.

[Ost02] K Ostermann. Dynamically composable collaborations with delegation layers. *ECOOP*, 2002.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: a comprehensive step-by-step guide*. Artima Inc, 2008.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[pic13] Picocontainer, 2013.

[PN06] D.J. Pearce and J. Noble. Relationship aspects. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 75–86. ACM, 2006.

[PS04] R Pawlak and L Seinturier. JAC: an aspect?based distributed dynamic framework. *Software: Practice . . .* , 2004.

[RB10] Andrew Lee Rubinger and Bill Burke. *Enterprise JavaBeans 3.1*. O'Reilly, 2010.

[RBL+90] James R Rumbaugh, Michael R Blaha, William Lorensen, Frederick Eddy, and William Premerlani. Object-oriented modeling and design. 1990.

[RHB13] Arnout Roemers, Kardelen Hatun, and Christoph Bockisch. An adapter-aware, non-intrusive dependency injection framework for java. In *PPPJ*, pages 57–66, 2013.

[RS03a] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[RS03b] Hridesh Rajan and Kevin Sullivan. Need for instance level aspect language with rich pointcut language. *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, 2003.

[SMU+06] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matuura, and S. Komiya. Design and implementation of an aspect instantiation mechanism. *Transactions on aspect-oriented software development I*, pages 259–292, 2006.

[Spa00] Michael Sparling. Lessons learned through six years of component-based development. *Commun. ACM*, 43(10):47–53, October 2000.

[SV13] Davy Suvée and Wim Vanderperren. Jasco website, 2013.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM, 2003.

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[TOHSJ99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999.

[Van08] Robbie Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework*. Apress, 2008.

[Vin97] Steve Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[Weg96] Peter Wegner. Interoperability. *ACM Computing Surveys*, 28:285–287, 1996.

[YBA13] Haihan Yin, Christoph Bockisch, and Mehmet Aksit. A pointcut language for setting advanced breakpoints. In *Proceedings of Aspect-Oriented Software Development*. ACM, 2013.

[YBB99] Daniil Yakimovich, James M Bieman, and Victor R Basili. Software architecture classification for estimating the cost of cots integration. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 296–302. IEEE, 1999.

[YCS+02] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 85–92, New York, NY, USA, 2002. ACM.