Abstraction over Non-Local Object Information in Aspect-Oriented Programming Using Path Expression Pointcuts

A Case of Object Persistence

Dissertation

zur Erlangung des akademischen Grades eines

Doktor der Naturwissenschaften
(Dr. rer. nat.)

durch den Fachbereich Wirtschaftswissenschaften der

Universität Duisburg-Essen
Campus Essen

vorgelegt von

Mohammed Ali Nagi Al-Mansari

geboren in Taiz, Jemen

Essen (Februar, 2008)

Tag der mündlichen Prüfung: **Freitag den 25.04.2008**

Erstgutachter: **Prof. Dr. Rainer Unland**

Zweitgutachter: **Prof. Dr. Cherif Branki**

In the memory of my father,
To my loving mother and sister,
To Nawal,
To Rafid, Raghid and Rifad
I love you all

Acknowledgments

I would like to take this opportunity to express my sincere gratitude to all the people who helped me in various ways in completing this study. Without each and every one of them, the completion of this thesis would have been a more arduous task.

First of all, my special thanks are due to my supervisor Prof. Dr. Rainer Unland for his insightful comments and constructive suggestions throughout the thesis. I thank him for his sound guidance, constant support and unfailing patience during the writing of this thesis. I am grateful to him because he displayed confidence in my ability to conduct meaningful research. He has contributed a lot to my growth. He offered me continued professional, financial and personal support during my study. I like him as a great scholar and as a wonderful human being.

I am indebted to Prof. Dr. Cherif Branki for all his generosity and academic help at different stages of my research. He helped me in making the thesis in a very good shape as well as enhancing the completion of the research in the specific time span.

It is a great pleasure to extend my deepest thanks to Dr. Stefan Hanenberg. It is with him that I developed my knowledge base and research interest in aspect-orientation and programming languages in general. His academic talent, unique enthusiasm, extraordinary patience, and constant encouragement significantly contributed to make my work a rewarding academic and personal experience. I am so obliged to him in the real sense of the term because he did not spare any effort to help me in the completion of this thesis, and without which this research would not see the light. I like in him the real human being and the friend that I will never ever forget.

I extend my gratitude to many good friends of mine who helped me, stood by me, and were always willing to offer their help and emotional support in so many different ways: Mosbah, Awny, Tarek, Esra, Raed and Tawfiq. A special heartfelt

"thank you" goes to my closest friends who are almost a part of my family. In particular, I want to thank Dominik Stein, Frank Büscher, and Oliver Horn for trusting me as they always have and for teaching me that 'ripeness is all'. Special thanks go to all of them because they gave me both moral and intellectual support during my stay in Essen. My great thanks also go to the ones who work behind the scene and without them, my work would be so difficult: Veronika Muntoni and Gottfried Merkel. Many thanks to Christian Gutmann and Pablo Sánchez for helping me fighting with $\LaTeX$.

Finally, I want to thank my mother, brothers, sister, wife and kids (Rafid, Raghid and Rifad) who always stood by my side with unconditional love and unshakable trust. They have provided me with intellectual and emotional support, encouragement and putting up with my absence since I began this research.

Abstract

Aspect-oriented software development (AOSD) consists of a number of technologies that promise a better level of modularization of concerns that cannot be separated in individual modules by using conventional techniques. Aspect-oriented programming (AOP) is one of these technologies. It allows the modularization at the level of software application code. It provides programmers with means to quantify over specific points in the base application code, called join points, at which the crosscutting concern code must be triggered. The quantification is achieved by special selection constructs called pointcuts, while the triggered code that is responsible for adapting the selected join point is provided by special construct called advice.

The selection and adaptation mechanisms in aspect-oriented programming depend heavily on the distinguishing properties of the join points. These properties can either be derived from the local execution context at the join point or they are considered to be non-local to the join point. Aspect-oriented systems provide a plenty of pointcut constructs that support accessing the local join point properties, while they rarely support the non-local properties.

A large research effort has been achieved to extend current aspect-oriented systems in order to solve the problem of non-locality. However, none of these proposals support the non-local object relationships. There are many situations where a good abstraction over non-local object information is needed, otherwise, the developers will be obliged to provide complex and error-prone workarounds inside advice body that conceptually do not reflect the semantics of join point selection and mix it with the semantics of join point adaptation. Such recurrent situations occur when trying to modularize the object persistence concern.

Object persistence, the process of storing and retrieving objects to and from the datastore, is a classical example of crosscutting concern. Orthogonal object persistence meets the obliviousness property of AOP: The base code should not be prepared upfront for persistence.

This thesis addresses the shortcomings in current aspect-oriented persistence systems. It shows that the reason for such shortcomings is due to the lack of supporting non-local object information by the used aspect-oriented languages.

To overcome this problem, this thesis proposes a new extension to the current pointcut languages called *path expression pointcuts* that operate on object graphs and make relevant object information available to the aspects. As an explicit and complete construct, a formal semantics and type system have provided. Moreover, an implementation of path expression pointcuts is discussed in the thesis along with its usage to show how the aforementioned problems are resolved.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programming for separation of concerns aims to provide better modularization for concerns that cut across several modules in the software systems. Within *aspect-oriented software development* (AOSD) a number of technologies emerge to allow multiple concerns to be separately expressed, but nevertheless be automatically unified into working systems.

One of these technologies is *aspect-oriented programming* (AOP). It provides features that allow the developers to quantify over specific parts of the base application code and its execution context. These parts are called *join points*. The quantification is achieved by means of the so-called *join point mechanism*, which uses special selection constructs called *pointcuts*. AOP provides other means by which the developers can specify what to do at the selected join points by using another feature called *join point adaptation*. Such adaptation is encapsulated in a special construct called *advice*.

Each join point has its own distinguishing characteristics and properties that differentiate it from other join points. Pointcuts must be able to access the join point properties in order to determine whether to select it. Similarly, in order to adapt a selected join point, it is necessary for the advice to get access to specific join point properties. Therefore, aspect-oriented languages provide mechanisms to expose the required join point properties from the join point context to the aspect context. This is called *context exposure mechanism*. It is provided by means of special pointcut constructs that are often called *binder* pointcuts.

A join point property is said to be *local* if it can be derived from the local execution context at the join point. On the other hand, a *non-local* property of a join point is the relevant information that can not be accessed directly from the local execution context of the join point.

Accessing local join point properties can be relatively easily implemented. Current aspect-oriented languages and systems typically support a large variety of such local properties. However, these systems are quite miserly with non-local join point properties. This is due to the fact that the provision of non-local join point properties is typically much more complex: The aspect-oriented system's task is to collect some data at runtime in order to use it at a later point in time for evaluating pointcuts. This is why a large research effort has been done to extend current pointcut languages to provide aspects with access to different kinds of non-local join point properties.

However, none of these proposals provide abstractions over the non-local properties that are based on object information, which in many situations, are required for join point selection and adaptation.

Hence, this thesis discusses a number of examples of such situations in the domain of object persistence, which is considered to be a classical example of a crosscutting concern. These examples illustrate how object persistence systems depend heavily on the object relationships.

The discussion covers a number of proposed aspect-based solutions that claim to provide orthogonal object persistence. Orthogonal persistence requires that the application code must be orthogonal to its persistence at the type-level and the code-level as well as it must comply with persistence by reachability.

The main contribution of the thesis is to assess to what extent current aspect-oriented persistence systems comply with the principle of orthogonal persistence, to reveal any shortcomings, and to provide solutions for these shortcomings.

The evaluation carried out in this thesis shows that, despite the available aspect-oriented persistence systems provide better level of orthogonality than the conventional persistence frameworks, they still compromise persistence orthogonality. Using some examples of object persistence, the thesis illustrates how these systems break the orthogonality at the type level as well as at the code level. Moreover, it shows how these systems break the principle of persistence by reachability.

Then, this thesis concludes that these shortcomings are mainly caused by the lack of good abstractions over the non-local object information in aspect-oriented programming that preserve the expressiveness of pointcut languages.

As a solution, this thesis proposes the *path expression pointcut* (PEP) as an explicit extension to the current pointcut languages. It operates on object graphs to provide access to the matching part of the graphs according to a given path expression pattern. PEP extends the mechanisms of parameter binding and context exposure. As a consequence, PEP also has its own semantics of advice execution.

This extension shifts the complexity from the level of coding to the level of programming language. Therefore, a clear understanding of the PEP becomes a must, e.g. in order to integrate it with the existing pointcut languages. In order to solve this problem, the thesis provides an unambiguous denotational semantics for PEP.

Since PEP passes a part of the object graph to the advice, which in turn must be able to process the objects that are part of the exposed graph, it is needed to provide a suitable interface for this graph. As most aspect-oriented languages are based on typed languages, it is required to provide a type-safety of the graph interface, which is also a part of this thesis.

Finally, a prototype implementation of PEP is given to prove the feasibility of the implementation of this complex construct. This implementation is used to show how PEP solves the problems described above.

1.1 Chapter Summaries

The thesis is organized as follows:

Chapter 2: Background. This chapter gives an introduction to the main concepts of the thesis work domain. It starts by defining the concept of the separation of concerns and the notion of crosscutting concern. Then, it introduces the technologies used in aspect-oriented software development with a focus on aspect-oriented programming. As an example of an AOP language, an introduction to AspectJ is given thereafter. The problem that is addressed in this thesis is from the domain of object persistence. This chapter defines this domain and a number of issues that must be considered to provide orthogonal object persistence. This discussion also covers a number of current conventional object-oriented solutions for object persistence as well as counterpart aspect-oriented solutions for object persistence.

Chapter 3: Problem Description. In this thesis, the first premise of orthogonal object persistence is that the persistence system must comply with the orthogonality principles: Type orthogonality, persistence independence, and transitivity. From an aspect-oriented programming perspective, the orthogonality requirement meets the obliviousness property of aspect-oriented programming.

This chapter presents a number of examples that show how current aspect-oriented persistence systems fail to fulfill the obliviousness characteristic of aspect-oriented programming, in other words, these systems compromise the orthogonal persistence principles. Using some examples, the thesis justifies this conclusion. These examples are related to pure persistence operations as well as to the concurrency control that is considered as a related issue to persistence. This illustration insists in the importance of non-local object relationships, e.g. reachability or even certain parts of the object graph, to aspectize object persistence, especially for update, delete, and retrieve operations. Another example illustrates this fact in a different situation when aspectizing other crosscutting concerns, e.g. observer design pattern. The chapter ends by stating the problem concisely: Current aspect-oriented systems do not support non-local object information.

Chapter 4: Path Expression Pointcuts. Following the problem statement, this chapter proposes a pointcut construct that applies the well known *path expressions* technique in aspect-oriented programming. The new pointcut is called *path expression pointcut* (PEP). The new construct is introduced informally: Its concrete syntax, semantics, pattern matching, parameter bindings and context exposure, how it modifies the mechanism of advice execution, and how it makes the non-local relevant information to the join point available for the aspect. The chapter then discusses how such informal description of PEP may result in an ambiguous understanding of the construct and its results. Moreover, the chapter motivates the need to provide a type system for the PEP in order to ensure its implementation in any typed aspect-oriented language.

Chapter 5: Formal Semantics and Type System. This chapter, first, proposes an unambiguous denotational semantics for the path expression pointcut. The derivation of the system starts from defining a mathematical model for object graphs, which is the context of PEP. Then, the formal semantics is presented along with examples that show how the formal semantics provides a clear understanding of the PEP results. A simple type system for PEP is provided thereafter.

Chapter 6: Implementation. This chapter presents a prototype implementation of PEP in a typed aspect-oriented language called PePAL. The discussion covers different facets of the implementation and provides examples that illustrate the usage of the construct.

Chapter 7: Motivating Examples Revisited. Using PEP in PePAL, this chapter solves the problems mentioned in Chapter 3.

Chapter 8: Related Work. This section discusses the related work to this thesis. The related work is divided into several domains: Conventional solutions for object persistence, aspect-oriented solutions for object persistence, non-locality issues in aspect-oriented programming, expressiveness of pointcut languages, path expressions and their applications, path expressions in aspect-oriented programming, formal semantics and type systems for path expressions and object graph, and finally formal semantics and type systems for aspect-oriented programming languages.

Chapter 10: Discussion and Conclusion. This chapter presents a discussion of the proposed solution. Then, it gives a summary of future work discussion and a discussion of the main contributions of the thesis. Then, it concludes this thesis by a summary of the work that is done by each chapter.

Chapter 2

BACKGROUND

This chapter introduces the domain of the thesis's work. Section 2.1 discusses the *separation of concerns* concept along with its importance to the field of software engineering. The notion of *crosscutting concern* is also going to be introduced.

Section 2.2 gives an introduction to the aspect-oriented software development in general with a focus on aspect-oriented programming. A number of different features, terminologies, and techniques that characterize the concepts of aspect-oriented programming will be discussed, e.g., quantification, obliviousness, join points, and weaving mechanisms. As an example of an aspect-oriented programming language, an introduction to AspectJ will be given in this section.

Object persistence service will then be discussed in Section 8.1. This will include the definition of *orthogonal persistence* in Section 2.3.1 and its three principles that must be fulfilled in order to achieve it. In addition to that, Section 2.3.2 discusses some issues that are related to object persistence that are going to be used in the thesis.

A number of current conventional object-oriented solutions for object persistence are explored in Section 2.4. Similarly, Section 2.5 gives a general overview of how current aspect-oriented programming attempts to provide corresponding solutions to object persistence.

Finally, a chapter summary is presented in Section 2.6.

2.1 Separation of Concerns

In software engineering, a concern is a general term that describes a piece of interest in a system and typically has its own features and behavior. It is desirable to isolate each concern as a unique conceptual unit, consequently, as a separated implementation module.

Separation of concerns is a term advocated by Dijkstra (1976):

> "...to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects [...]. It is what I sometimes have called the *separation of concerns*."

In other words, it means to break down a problem into easier, more understandable, manageable, and maintainable subproblems that could be solved individually.

Some researchers refer to this process as the *software system decomposition* [Courtois (1985); Parnas (1972)]. However, this thesis is making use of both terms interchangeably to describe the same concept. Moreover, it considers the term *modularity* [Parnas (1972)] as being synonymous to the term separation of concerns.

During each phase of the process of software development, separation of concerns could be described, however, in different levels of flexibility. For example, in the analysis and design phase, the separation of concerns is described informally as human readable documents that are easy to understand and managed due to the fact that this phase relies on natural language. On the other hand, the implementation phase is achieved by using programming languages that have a set of formal rules to be met, so the programs can be interpreted and run by the computer.

The aim of the programming language design is to provide powerful syntactic and semantic constructs that allow the programmers to specify an implementation that is as closely as possible to the analysis and design of a problem, which represents the mental model of the programmers [Stein et al. (2006)]. By means of such expressive constructs the programmers will be able to specify maximal separation of concerns at the implementation level.

There are many benefits in applying the separation of concerns to a program construction. First of all, the program that solves a problem by dividing it into pieces is readable and understandable by humans also in pieces. Second, since concerns are localized in separate modules of the program, any change in the requirements of one concern is localized only to the implementation code of that concern. Another benefit is concerned with the maintenance of the program. If there is an error in the functionality of a single concern, it is easier to detect the error and maintain this functionality in a localized implementation code of that single concern. Moreover, with suitable mechanisms, the concerns could be integrated into programs. Last but not least, a good modularization of a given concern provides the ability to reuse this concern in other applications.

The term *crosscutting concern* [Kiczales et al. (1997)] is used to describe a concern that is scattered and tangled amongst other concerns so that it cannot be well modularized. This is a result not only from the nature of the concern, but also from the available abstraction mechanisms that are used in the separation of concerns [Hanenberg (2005)].

Each programming system that offers techniques to decompose the problem into separated modules on the implementation level provides means to bring together these separated modules of this problem. Such techniques should guarantee during the program development process that these different modules fit together rather than being surprised during the execution of the program by discovered incompatibilities.

Finally, it must be stressed that this thesis will consider the separation of concerns at the level of the implementation code of these concerns and not in the early stages of the program development process.

2.2 Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) is an approach to software development that addresses crosscutting concerns. It provides a set of emerging technologies that are used to encapsulate crosscutting concerns in separate modules, known as aspects, so that localization can be promoted [Filman et al. (2005)]. This results in better support for modularization hence reducing development, maintenance, and evolution costs. Some examples of crosscutting concerns addressed by AOSD are logging, security, persistence, memory management, failure handling, debugging, and synchronization.

Aspect-oriented (AO) approaches offer also the modularization at the different stages of the software lifecycle. These levels can range from the problem space, e.g., requirement engineering, analysis, and modeling, to the solution space, e.g. implementation, coding, and testing.

In fact, AOSD is a rapidly evolving area. As a consequence, there is a large effort of work that illustrates the applicability of AOSD technologies for software modularity. This is applied to the early stages of a software development process, e.g., requirement engineering [Chitchyan et al. (2005)], architecture and design [Jackson et al. (2006); Krechetov et al. (2006)], and modeling [Hanenberg et al. (2005); Stein et al. (2004)]. However, as stated before, this thesis will consider the software modularity at the level of implementation. Hence, early stages of separation of concerns with AOSD will not be discussed further.

2.2.1 Aspect-Oriented Programming

The term *aspect-oriented programming* (AOP) was introduced by Kiczales et al. (1997). It is used to describe the application of programming in the implementation of crosscutting concerns. In other words, it addresses the problem of crosscutting code. Aspect-oriented programming is applied to different types of programming paradigms in order to solve the lack of good modularity. For example, there are many aspect-oriented programming languages that could be used in combination with corresponding object-oriented programming languages such as AspectJ [Kiczales et al. (2001a,b)] for Java [Gosling et al. (1996)], AspectS [Hirschfeld (2003)] for Smalltalk [Goldberg (1984)], and AspectC++ [ACPP (2007); Spinczyk et al. (2002)] the aspect-oriented extension to C++ [Stroustrup (1997)].

Aspect-oriented programming offers a set of techniques that permit to address the code that cuts across multiple modules and encapsulate it in separate modules called aspects. From this point on, the term aspect is used to describe the separated concern, while the term base application is used to describe the rest of the modules. Similar to other modularity techniques, the aspect-oriented techniques provide a means to compose these modules together into a working system [Filman et al. (2005)].

First of all, as mentioned above, the crosscutting concerns are modularized in so-called aspects. Similar to the class in object-oriented programming, the aspect contains the specifications that define the crosscutting concern by using well-defined constructs. *Quantification* and *obliviousness* have been introduced by Filman and Friedman (2000) as the two distinguishing characteristics of aspect-oriented programming. Quantification means the ability to specify aspects that influence multiple program modules. Obliviousness on the other hand means that those base modules are not prepared to be quantified by those aspects.

The idea of quantification in aspect-oriented programming is the ability to select specific points in the program in order to affect them. These points are called *join points*. The term join point was introduced by Kiczales et al. (2001a,b): "Join points are principled points in the execution of a program". A method call, a method execution, and a field access are some examples of join points.

Each join point has its own distinguishing characteristics that make it different from other join points. These characteristics are called either *join point properties* [Hanenberg (2005)] or *predicates* [Gray et al. (2003); Gybels and Brichau (2003)]. This thesis is making use of the term join point properties. There are different kinds of join point properties, e.g., method names and return types of method call join points.

Since join point properties are important in distinguishing the join points from each other, Hanenberg (2005) investigated the factors that influence the join point properties as one of the design dimensions of aspect-oriented systems. In that work, the author divides these factors into four categories:

Dynamicity. Whether a join point property can be derived from the available static information (code) or this requires some dynamic information at run-time.

Directness of property correspondence. A join point property has a direct correspondence, if it represents some data that is available directly at the join point. If it needs additional computation from the aspect-oriented system to obtain a property at a given join point, this property is said to have an abstract correspondence.

Locality. When a join point property could be derived directly from the local context information of this join point, this property is said to be *local* to the join point, otherwise, it is considered as a *non-local* join point property.

Application progress. It distinguishes between three states of a dynamic data property at the current join point. That is, whether the data property represents data from the previous, current, or future system state.

The thesis will consider the locality issues of the join point properties. Later on in Section 2.5, a more detailed discussion about this issue is presented. Moreover, the next chapter discusses why locality of join point properties is so important in aspect-oriented programming in order to provide better aspect-based modularization.

Typically, there are two mechanisms that are applied to the join points, namely, *join point selection* and *join point adaptation*. The selection of the join points is performed by using a set of selection predicates or criteria called *pointcuts* [Kiczales et al. (2001a,b)]. The term *pointcut language* is mostly used in the aspect-oriented literature to describe the set of pointcut designators in a given aspect-oriented system, e.g. [Ostermann et al. (2005)]. The join point adaptation is done by the so-called *advice mechanisms* [Kiczales et al. (2001a,b)]. Inside advice body, developers are able to add an additional code that is used to add specific behavior normally represented by the aspect at the selected join point from the base code.

There is also another type of adaptation of the base program, which is about to affect the structure of its modules, which is called *inter-type declarations* or *introductions*. This kind of adaptation does not affect the join points, rather it is used to add new elements to the modules, e.g., in AspectJ, to add new members (e.g. fields and methods) to the classes or to modify the inheritance hierarchy of these classes.

Aspect-oriented programming also provides a mechanism to compose the separated concerns it produces with all other modules of the application. This is achieved by the so-called *weaving mechanism* [Hilsdale and Hugunin (2004); Kiczales et al. (1997)]. There are two kinds of weaving, *static weaving* and *dynamic weaving.* In the static weaving, aspects are integrated with the applications at compile-time, e.g., weaving in AspectJ. In the dynamic weaving, e.g., in AspectS [Hirschfeld (2003)] and PROSE [Popovici et al. (2002)], the integration of the aspects and the base code is determined dynamically.

In what follows, a brief introduction to the programming language AspectJ is presented. AspectJ is considered to be the most dominant and mature aspect-oriented programming language up to now. The purpose of this introduction is to elaborate the meaning of the above mentioned terminologies and concepts. Moreover, most of the examples throughout the thesis will be illustrated by using AspectJ code.

2.2.2 An Introduction to AspectJ

AspectJ [Kiczales et al. (1997, 2001a,b); Laddad (2003)] was developed by Xerox Palo Alto Research Center in 1997. Recently, it becomes part of the eclipse project [ASPJ (2007)]. AspectJ is based on the programming language Java where its compiler is an extension to the Java compiler. It follows the structure and the syntax rules of Java. The aspects in AspectJ have the shape of Java classes with members like fields and methods. Moreover, the AspectJ compiler can compile a pure Java code.

AspectJ is used intensely in the different research fields of aspect-oriented programming. A large number of proposals have been published in order to add new semantics to AspectJ, e.g. [Harbulot and Gurd (2006); Sakurai et al. (2004)], or to prove the need for new language semantics by using AspectJ, e.g. [Hanenberg and Unland (2003); Hanenberg et al. (2004); Kienzle and Guerraoui (2002); Soares et al. (2002)]. For this reason, Avgustinov et al. (2005) introduced the so-called AspectJ bench compiler, *abc* for short, a workbench to make it easier for researchers to develop AspectJ language extensions as well as compiler optimizations.

AspectJ fulfills the requirements for aspect-oriented programming discussed in the previous section by providing the following constructs (in addition to the ordinary Java language constructs): Pointcuts, advice, inter-type declarations, and aspects.

2.2.2.1 Pointcuts

Pointcuts are the constructs used to select certain join points in the application base code. There are different kinds of join points that are selected by corresponding pointcuts. Those kinds are: method call, method execution, exception handling, instantiation, constructor execution, and field access. Each kind of join point can be picked out by its own specialized pointcut. For example, the `call` pointcut is used to select method call join points, and the `set` pointcut is used to select field set join points.

In addition to matching join points, some pointcuts in AspectJ are used to expose the context from the selected join points to the aspects. The operation of exposing the join point context is performed by the so-called *context exposure mechanism* [Chiba and Nakagawa (2004); Hilsdale and Hugunin (2004)]. For example, the `target` pointcut designator is used to expose the target object of the matching join point. In this thesis, the terms pointcut and pointcut designator are used interchangeably.

Pointcuts are either named or anonymous and can also be grouped together to form more complex pointcuts. A named pointcut is defined by the keyword `pointcut` followed by an identifier that can be used to refer to it. A named pointcut can be defined as a member of an aspect, an interface, or a class. Anonymous pointcuts are defined directly at advice, as an argument to some pointcuts such as `cflow` or as a part of other pointcut definitions. Pointcut designators could be combined by using the operators “`&&`”, “`||`”, or “`!`” in order to compose more complex pointcuts.

For example, the pointcut named `figureMoved` in Figure 2.1 picks out each method call join point where the method name has a prefix `set` and the target of this method is an object of type `FigureElement`. AspectJ permits specifying formal parameters in the pointcut headers, e.g. `fe`, but these parameters must be bound in the rest of the pointcut definition to corresponding objects from the join point context. Here, the `target` pointcut bounds the target object of the method call join point to `fe` and then exposes it to the aspect.

```
pointcut figureMoved(FigureElement fe):
    call(* FigureElement+.set*(..)) && target(fe);
```

Figure 2.1: A pointcut definition in AspectJ

Another observation from the example above is that the method signature in the call pointcut designator is making use of the wildcard "`*`" twice: First, it is used to indicate that the method could have any access modifier and return type. Second, it is used to indicate that the matching method could have any suffix. Another wildcard "`+`" is used in the type pattern `FigureElement+` in the method signature to indicate that this method is a member of the type `FigureElement` or any of its subtypes. The last wildcard is "`..`", which means that the matching method could have zero or more arguments.

A complete discussion about AspectJ's pointcut language is out of the scope of the thesis. The complete list of AspectJ pointcuts, their syntax, and semantics is available in [ASPJ (2007)].

2.2.2.2 Advice

The advice is a construct that permits AspectJ developers to define aspect behavior and additional code to be performed at the selected join points by a pointcut. Its definition is similar to a method declaration. However, it is not possible to invoke the advice explicitly; rather, its invocation is added implicitly to the base code by the AspectJ compiler.

There are three kinds of advice: `before`, `after` and `around`. The execution of `before` advice takes place prior to the selected join point. The `after` advice can be used either in the general non-restricted form, in the restricted returning form, or in the throwing form. In these forms, the advice body is executed after the execution of the join point, after returning from the join point, or after throwing an Exception from the join point. The around advice execution surrounds the join point, where it could be executed as a before or an after advice. Moreover, around advice can bypass or proceed the execution of the join point with the same or different context by means of the proceed expression.

The advice is associated with a pointcut and has the same parameters as those specified by the associated pointcut. Moreover, the advice has a body similar to the body of the method.

```
after (FigureElement fe): figureMoved(fe) {
    fe.reDraw();
}
```

Figure 2.2: An after advice making use of figureMoved pointcut

As an example of an advice declaration, Figure 2.2 shows an after advice that is making use of the predefined pointcut figureMoved in Figure 2.1 shown above. The advice has the same formal parameter as the pointcut, which means that the corresponding bounded FigureElement object is exposed from the join point context and is made available for the advice to use it. Inside the advice body, the method reDraw is being invoked on the object fe.

2.2.2.3 Inter-type declarations

Inter-type declarations, which are used to be called introductions, are constructs to declare new members, fields or methods, to classes or to change the inheritance hierarchy of the existing base code types by adding new supertypes to them. Hence, this kind of base code modification is used to change the static type structure of the base code. The declaration of the new members to the classes is accomplished inside the aspects that are associated with these classes. The extends and implements relationships are defined by using the construct declare parents inside the aspect.

Figure 2.3 illustrates both kinds of inter-type declarations. Inside the aspect Coloring, an interface called Colored is defined, then two members are added to it, i.e., the field `myColor` of type `Color` and the private method `colorMe` to mutate that field. Then, the aspect declares the `FigureElement` class as an implementer of the `Colored` interface. It must be noted that AspectJ allows to introduce non-final private fields to the interfaces as well as introducing non-public and non-abstract methods to the interfaces.

```
aspect Coloring {
    interface Colored {};
    private Color Colored.myColor;
    private void Colored.colorMe(Color c) {
        myColor = c;
    }
  declare parents: FigureElement implements Colored;
}
```

Figure 2.3: Inter-type declarations in AspectJ

2.2.2.4 Aspect

The `aspect` is a modular container for pointcuts, advice, and inter-type declarations. It is defined much like a class, and it may contain any class member declaration. It is also possible to create abstract aspects, which can contain abstract pointcuts and abstract methods. Aspects may extend each other or implement interfaces. Figure 2.4 shows how a simple aspect called `Coloring` is defined.

Unlike classes, the developers are not allowed to instantiate aspects. The compiler is responsible for the instantiation of the aspect. Moreover, the aspects are singleton by default. Hence, only one aspect instance is created. However, it is possible to associate aspect instances to objects and control flows by using `perthis`, `pertarget`, `percflow`, and `percflowbelow` clauses [ASPJ (2007)].

2.2.2.5 Aspect Weaving

The integration of aspects and the application code is achieved by means of the *weaving process*. In AspectJ, the weaving is done at compile time. The advice

```
aspect Coloring {
    interface Colored {};
    private Color Colored.myColor;
    private void Colored.colorMe(Color c) {
          myColor = c;
    }

    declare parents: FigureElement implements Colored;

    pointcut figureMoved(FigureElement fe):
          call(* FigureElement+.set*(..)) && target(fe);

    after (FigureElement fe): figureMoved(fe) {
          fe.colorMe(new Color(0x00, 0xff, 0x00));
          fe.reDraw();
    }
}
```

Figure 2.4: A complete aspect definition in AspectJ

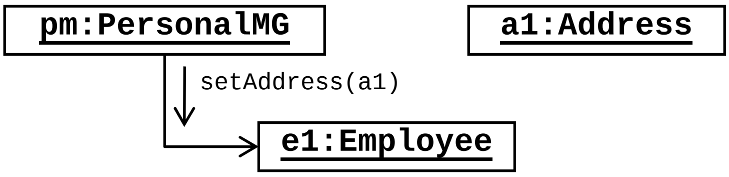
body is added to the base code at every potential matching join point to its associated pointcut [Hilsdale and Hugunin (2004)]. These join points are matched statically based on the so-called *join points shadows*.

Some of these potential join points need some dynamic information to determine whether to apply the woven code or not. This type of join points is called *dynamic join points*, e.g. those join points involved in `cflow` or `target` pointcuts. The *static join points*, on the other hand, are those which are fully matched using the static information, e.g. by using `within` pointcuts.

2.2.2.6 Parameter Bindings and Context Exposure in AspectJ

One feature of pointcuts in languages such as AspectJ is the context exposure mechanism [Chiba and Nakagawa (2004); Hilsdale and Hugunin (2004)]. Context exposure in AspectJ permits to pass objects from a pointcut described by the dynamic pointcuts `this`, `target` or `args` to the pointcut and then to the advice. This decouples the join point selection from the advice and permits different pieces of advice to access join point properties in a unique way. Some kinds of join point properties, e.g., signature patterns in AspectJ, cannot be exposed by the context exposure mechanism.

The context exposure mechanism starts by collecting the objects related to the pointcut. These objects are specified by corresponding variables in the context exposure based pointcuts, i.e. `this`, `target` and `args`. For example, according to the collaborative diagram of Figure 2.5, the `PersonalMG` object `pm` is trying to assign the `Address` object `a1` to the address field of the `Employee` object `e1` by means of invoking the method `setAddress`.



```
pointcut changeAddr(Employee emp, Address addr):
  execution(* *.*(..)) && args(addr) && target(emp);
```

Figure 2.5: Collaborative diagram of a method invoked to modify an `Employee` object

A possible pointcut that captures this method execution join point might be defined as shown in Figure 2.5, which is called `changeAddr`.

AspectJ restricts each formal parameter in the pointcut header to be bound in the body of the pointcut with exactly one designator. Here, `emp` is bound to the target of the method execution join point, which is the object `e1` as the diagram shows. This is done by using the `target` pointcut designator. Similarly, the `args` pointcut designator binds the formal parameter `addr` to the object `a1`. The binding set `(emp=e1, addr=a1)` is going to be part of the available context within this pointcut that can be accessed directly from the advice.

2.3 Object Persistence

Object persistence is the ability of objects to survive the termination of the program that created them. This is achieved by storing the objects in nonvolatile storage, so if the application is terminated, other applications can get these objects from this storage. Such storage is called persistent storage or datastore, e.g. file systems and database systems. Objects that die by the end of a program are called transient.

Persistence is one of the most considerable crosscutting concerns that are involved in most of nowadays software applications. It has been shown that persistence related code acquires typically 30% of the total application code [Atkinson et al. (1983b); King (1978)]. This code is concerned with transferring objects to and from files or database systems. This has a negative impact on the developer by distracting him from implementing her/his application logic.

There are different techniques that are used to apply object persistence (cf. Kemper and Moerkotte (1994)). First, persistent types are declared upfront, so that every object of a given persistent type will be made persistent. Another approach is to select each persistent object explicitly within the application. Most of the persistence systems designate persistent objects if they are referenced by another persistent object [Atkinson et al. (1983a)]. In addition to that, both of these techniques can be used together in the same system.

2.3.1 Orthogonal Persistence

Accordingly, Atkinson et al. (1983a) introduced the concept of *orthogonal persistence*, which means that data objects are orthogonal to their persistence. In order to achieve orthogonal object persistence, three dominant principles have to be complied with [Atkinson (2000); Atkinson and Morrison (1995)]:

Persistence independence. The promotion of code reuse, i.e., the code should be applicable for both transient objects and persistence objects. This also indicates that the persistence framework semantics must not be changed in both cases. Hence, the developer is not concerned with writing a code to move objects to and from the datastore. This principle is also known as *transparent persistence.*

Type orthogonality. All objects can be persistent or transient irrespective of their types, sizes or any other property. This principle keeps the programmer from providing persistence support by hand for those data types the language lacks persistence support. Such handwork is undesirable since it distracts the programmer from her/his actual task, namely implementing the application logic.

Transitive persistence. This means to identifying the objects to be persistent if they are in the object closure of (referenced from) a persistent object. Many researchers from the database and programming languages communities refer to this principle as *persistence by reachability* [Atkinson et al. (1983a)]. Whenever data is stored, everything that is necessary to use that data correctly has to be stored as well. This principle prevents dangling references. Moreover, it assures that stored objects can be correctly interpreted upon retrieval since the principle is applied to objects and their classes likewise.

In summary, these principles play the main role in developing well-engineered persistence applications. The persistence independence principle guides to reusable persistence frameworks, the type orthogonality principle allows the developers to concentrate only on their application logic, and finally, the transitivity principle ensures the consistency of stored data.

The rest of this section discusses some important related issues of the object persistence concern. Then a number of available conventional techniques for object persistence are introduced. These issues as well as solutions are going to be used throughout the thesis.

2.3.2 Issues of Object Persistence

In order to support orthogonal object persistence in every system, there are some other related concerns that need to be addressed. The rationale is that software applications became large scaled, distributed and enterprise-oriented. As a consequence, more features influence the persistent data and the persistence system. For example, the database management systems and the persistence programming language and frameworks must preserve the consistency and availability constraints for the stored data [Atkinson (1992); Atkinson and Morrison (1995)]. A number of these issues are discussed in the following.

Transaction management. A mechanism that provides the requirements of *atomicity*, *consistency*, *isolation*, and *durability* (ACID) of the persistent objects [Elmasri and Navathe (2000); Gray (1981)]. Atomicity means that either *all* data operations that form a single transaction must be performed and committed or none. Consistency ensures that a transaction moves data from one consistence state to another. The isolation property ensures that there is no interference between concurrently executed transactions. Finally, durability means that all changes that are applied to the data by a committed transaction must persist in the database.

Persistence approaches such as *Hibernate* [Bauer and King (2005)] and *Java Data Objects* (JDO) [Jordan and Russell (2003); Roos (2002)] provide a transaction interface that is used by the developer to execute the transaction demarcation operations: `begin`, `commit` and `rollback`. *Enterprise Java Beans* (EJB) [Sarang et al. (2001); SUN (2006a)] also provide this feature in addition to the container-managed transaction feature, where the container is responsible for issuing the transaction demarcations (*begin*, *commit*, *rollback*, and *abort*).

Concurrency control. Mechanisms that support concurrent accesses to the persistent data while preserving the illusion that each user is executing alone on a dedicated system. These mechanisms are used to ensure the isolation property of concurrent transactions or threads. Concurrency control approaches are based on different set of protocols, such as data item locking, timestamps, multi-version, and validation techniques. The concurrency control approaches could be part of the persistence system (e.g. JDO) or part of the programming language (`synchronize` or `lock` packages in Java).

Distribution. The persistent data may be stored in different locations and also may be accessed by clients from different locations. Thus, the persistence system must provide means to manage the distributed persistent data as well as to allow distributed clients to access this data. For example, 3-tier systems such as J2EE application server [SUN (2003)] provide means to manage non-localized data like JavaServlet API [SUN (2003)] and to access it by means of *remote method invocations* RMI API [SUN (1999)].

Persistence operations (data synchronization). These operations are used to synchronize the state of the data between the datastore and the transient representation of this data. Data synchronization is known as CRUD operations that include the persistence operations *save* (*create*), *update*, *retrieve* and *delete*. Persistence programming languages offer the so called *query language*, which is used by the developers to perform the persistence operations, e.g. the query languages EJB-QL [SUN (2006a)] and JDO-QL [Jordan and Russell (2003)].

Object identity. Persistent objects require unique identities in order to uniquely identify them for storage and retrieval. Each object has a unique identity, which is independent of its current state. Object identity is about finding the same object in another system or loading the same object that was referenced before, and this is at the heart of all distributed systems and entity based containers. Persistence systems normally provide the ability to specify the object identity either by the system itself or by the application, e.g. JDO [Jordan and Russell (2003)].

However, it is not that easy to issue and maintain the identity of an object because it must be system-wide unique, cannot be changed during the object-lifetime, and cannot be reused after the deletion of the object. Moreover, the situation is more complex if the application is distributed between heterogeneous platforms because each machine issues a different object identity for the object it holds.

Exception handling. Exceptions are significant to any application whether it involves persistence operations or not. For persistence systems, a lot of situations should be managed carefully in order to avoid any failures, either software or hardware ones. For example, in Java based distribution operations, the remote method invocations (RMIs) are used and the application should handle the exceptions raised from those methods, namely `RemoteException` or some of its subclasses.

The point here is that the type of an exception differs from one operation to another, and hence, making use of a general exception pattern may help and is important to the persistence system. Most of the programming languages provide a mechanism to handle the exceptions in a form of `try-catch` block.

Datastore connectivity. Persistence systems provide means by which the applications can create the connections to the datastore. For example, if the application uses a database, every operation on a persistent data needs a connection between the application and the database. This connection, for consistency, security, and performance issues reasons, should be released when the operation is finished. In this case, the application may contain a lot of redundant code to manage the data store connectivity. Hence, it is important to make the manipulation of the data store connectivity transparent to the application and let the persistence systems manipulate these connections.

In general, all these issues must be addressed carefully in order to provide a complete object persistence solution. This thesis does not provide a complete persistence framework or language, rather, the main goal of the thesis is to point out some critical problems in the current persistence solutions in general and in the aspect-oriented ones in particular. Then the thesis will provide solutions that overcome these bottlenecks.

Consequently, this thesis will consider the modularization of some of the aforementioned persistence issues. Mainly, it will show how current persistence solutions (aspect or non-aspect based) modularize some well-known concurrency control policies and what these solutions still suffer from. Moreover, the modularization of persistence operations (save, retrieve, delete, and update) is also addressed throughout the thesis. However, other issues such as the transaction management and the distribution concerns are discussed briefly in some parts of the thesis whenever required.

2.4 Conventional Persistence Approaches

Many research efforts have geared towards adding orthogonal persistence to object-oriented programming or providing orthogonal persistence as separated tools and frameworks. These proposals were either research-oriented or have been implemented in real world systems. Based on Atkinson (2000), the object-oriented programming persistence approaches are categorized as the following subsections show.

2.4.1 Object Serialization

Object Serialization mechanisms are used to encode respectively decode object graphs into and from binary representations in a process called *serialization*. These mechanisms also support a complementary process called *deserialization* to reconstruct the object graph from its binary representation. Objects to be stored and retrieved frequently refer to other objects. When an object is stored, all of the objects that are reachable from that object are stored as well. Similarly, when an object is retrieved, all its referenced objects must be retrieved at the same time to maintain the relationships between the objects.

These mechanisms fail to support orthogonal persistence because of several reasons [Atkinson (2000)]. For example, object serializations are not orthogonal since, e.g., in Java, the types that are to be serialized must implement the `Serializable` interface. Therefore, the core classes and the classes that are imported as a byte code could not be made persistent because there is no way to ensure that those classes are implementing the `Serializable` interface. As a consequence, type orthogonality as well as the transitivity principles fail.

Moreover, these mechanisms do not scale well, e.g., retrieving a small portion of a big serialized data structure requires decoding of the whole data structure. Another point is that the object serialization fails to support persistence independence. For example, if two different objects are serialized and they share the same data structure, there would be two duplicate copies of this data structure after the deserialization. Hence, it does not preserve previously shared sub-structures.

2.4.2 Object-Relational Interfaces and Mapping

Mechanisms rely on a two-tiered architecture consist of an object-oriented programming language and a relational database management system. The programming language offers an application programming interface (API) that allows the communication between the language and the underlying database. For example, Java provides the *Java Database Connectivity* (JDBC) [Reese (1997)] to issue dynamic SQL statements and SQLJ [Clossman et al. (1998)] for statically checked embedded SQL statements.

However these mechanisms suffer from the *impedance mismatch* [Elmasri and Navathe (2000)] between the object model of the object-oriented programming language and the database relational model. The programmers are forced to maintain complex mappings between the two incompatible models, which breaks the transparency feature of orthogonal persistence. Although there are some object-relational mapping tools on the level of programming languages, e.g. JDO [Jordan and Russell (2003); Roos (2002)], these mechanisms require a previous knowledge of the mapping and result in complex mapping code and schemes that are not reusable.

2.4.3 Object Database Interface

A number of *object-oriented databases* (OODB) [Rao (1994)] have been introduced to follow the standards of the *Object Database Management Group* (ODMG) [Cattell et al. (2000)]. It defines the bindings to the object-oriented programming languages Smalltalk, C++ and Java. Examples of OODB are *GemStone* [Butterworth et al. (1991)], *Versant* [Wietrzyk and Orgun (1998)], *Objectivity/DB* [OBJ (2006)] and *ObjectStore* [Lamb et al. (1991)].

Object database interfaces do not suffer from the impedance mismatch as relational interfaces do. Apart from the easy mapping, however, programmers must control the persistence operations using these interfaces, which in turn compromises the principle of persistence independence.

2.4.4 Persistence Frameworks

Persistence frameworks support persistence in object-oriented programming either for managed or non-managed environments or both. *Managed environments* are environments whose objects are controlled by common application servers, e.g. the Sun Microsystem application server that is bundled with Java 2 Enterprise Edition (J2EE) [SUN (2003)]. Enterprise Java Beans (EJB) [Sarang et al. (2001);

SUN (2006a)], for instance, is a persistence framework that supports managed environments.

On the other hand, the applications that run in *non-managed environments* do not require a control of an application server and are composed of the so-called *plain old Java objects* (POJO) [Roos (2002)]. Such environments are supported by, for instance, Java Data Objects (JDO) [Jordan and Russell (2003); Roos (2002)] and Hibernate [Bauer and King (2005)].

The assessment in the motivation part of this thesis covers a number of the above technologies; namely, EJB, JDO and Hibernate. Hence, a brief introduction to these technologies is necessary.

2.4.4.1 Enterprise Java Beans

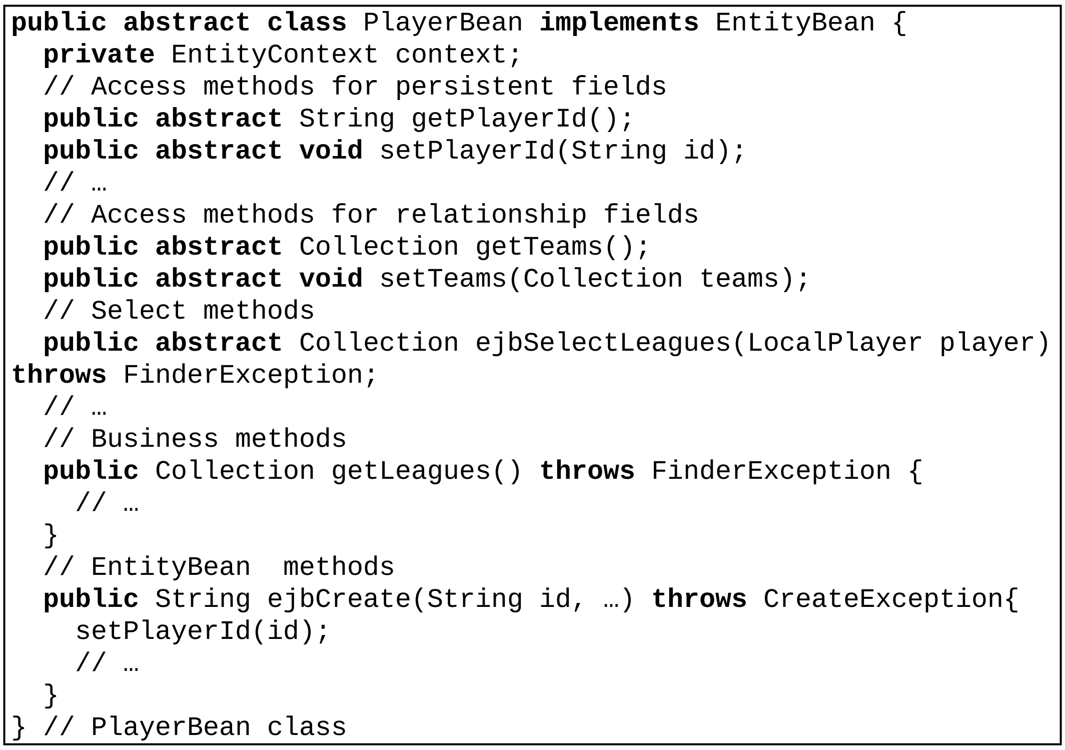
The Enterprise Java Beans (EJB) framework [Sarang et al. (2001); SUN (2006a)] aims to provide a complete solution to applications that manage distributed and heterogeneous data. The EJB infrastructure promises to automate many aspects of object persistence and transaction management. The EJB server also provides instance pooling and object-cache management. EJB comes as part of the J2EE application server.

There are three kinds of beans that could be defined: *Session*, *Entity*, and *Message-Driven* beans. A bean is a Java type whose objects cannot live outside their container. EJBs communicate with the outside world of their containers via a specific interface that is established by following certain rules.

With respect to persistent beans, there are two types, either *bean-managed persistence* (BMP) or *container-managed persistence* (CMP). In the bean-managed ones, the programmers need to take care of managing the persistence issues of the bean. The second type provides a sort of transparent persistence where persistence issues are manipulated by the container. This thesis considers this type of persistence management, which in most cases makes use of entity beans. Similarly, EJBs provide *bean-managed transactions* (BMT) and *container-managed transactions* (CMT).

In order to define a persistent type in EJB, the programmers define entity beans that must implement the `javax.ejb.EntityBean` interface. Moreover, two other interfaces should be defined: A remote interface extending the `javax.ejb.EJBObject` interface and a home interface that must extend the interface `javax.ejb.EJBHome`.

Accordingly, the programmers must follow certain naming conventions to identify the methods that affect persistent objects. For example, the setters and getters methods of a given property in the entity bean must have names of the forms `setProperty` and `getProperty`, respectively. Moreover, the select and finder methods follow similar naming conventions to allow the *EJB Query Language* (EJB-QL) to use them. The select methods should be `public` and `abstract`, throw the exception `java.ejb.FinderException` and should start with `ejbSelect`. The finder methods must be `public`, throw the exception `java.ejb.FinderException`, and should start with `find`.



```
public abstract class PlayerBean implements EntityBean {
  private EntityContext context;
  // Access methods for persistent fields
  public abstract String getPlayerId();
  public abstract void setPlayerId(String id);
  // …
  // Access methods for relationship fields
  public abstract Collection getTeams();
  public abstract void setTeams(Collection teams);
  // Select methods
  public abstract Collection ejbSelectLeagues(LocalPlayer player)
throws FinderException;
  // …
  // Business methods
  public Collection getLeagues() throws FinderException {
    // …
  }
  // EntityBean  methods
  public String ejbCreate(String id, …) throws CreateException{
    setPlayerId(id);
    // …
  }
} // PlayerBean class
```

Figure 2.6: Definition of the `PlayerBean`

For example, the `PlayerBean` is defined in Figure 2.6. It shows how the entity bean must extend the `EntityBean` interface to define the `ejbCreate` method and how it follows the naming conventions to define the access methods of the bean. The business methods also must follow these conventions. The select methods start always with `ejbSelect` and must throw the corresponding exception of finder methods.

```
public interface LocalPlayerHome extends EJBLocalHome {
  public LocalPlayer create(String id, …) throws CreateException;
  // Finder methods
  public LocalPlayer findByPrimaryKey(String id) throws
FinderException;
  // …
}
public interface LocalPlayer extends EJBLocalObject {
  public String getPlayerId();
  // …
}
```

Figure 2.7: Definition of the `Player` home and local interfaces

After that, the home and local interfaces must be defined for the `PlayerBean` in Figure 2.7. The finder methods must start with `findBy` and in the `LocalPlayer` interface the getter methods start with the prefix `get`.

2.4.4.2 Java Data Objects

The Java Data Objects (JDO) [Jordan and Russell (2003); Roos (2002)] framework attempts to provide persistence for ordinary Java objects. Such objects are called Pain-Old-Java-Objects (POJO). JDO has a much simpler infrastructure than the one of EJB. The JDO API provides access to a wide variety of heterogeneous data sources by using a simple OR-mapping system which is seen to a Java application as a part of the JDO source.

JDO instances implement the `PersistenceCapable` interface, either explicitly by the class writer, or defined by using the so-called *persistence descriptor* or *metadata* file. During the deployment of the JDO applications, the JDO enhancer modifies the byte-code of the plain objects according to the persistence descriptor in order to add persistence to specific types. Moreover, the OR-mapping system uses persistence descriptors to produce the database tables.

There is no need to apply certain naming conventions in defining accessor and mutator methods. In order to make objects persistent in the application code, the programmer must explicitly use the special method `makePersistent` of the `PersistenceManager` objects. JDO provides a sort of pool that can contain all `PersistenceManager` objects the programmer instantiated. Similarly, JDO provides an API for transaction management, which is also explicitly used by the programmer to instantiate and demarcate the transactions (begin, abort, and commit operations).

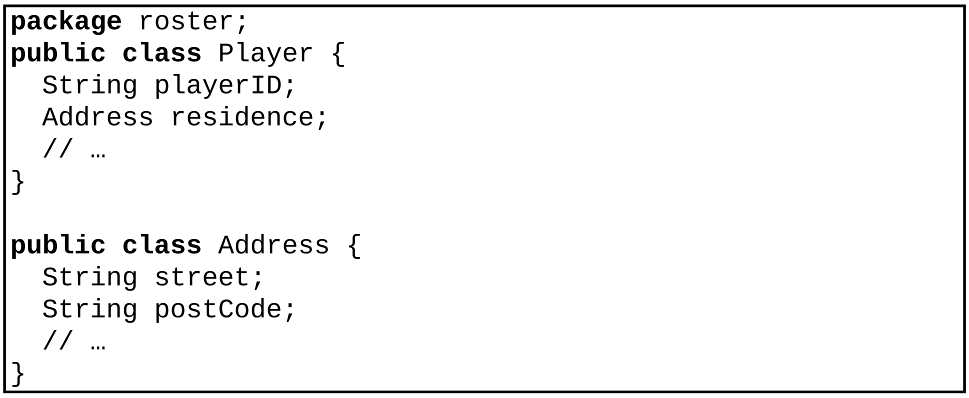
JDO provides three types of object identity:

1. Application identity - JDO identity managed by the application and enforced by the datastore; JDO identity is often called the primary key.

2. Datastore identity - JDO identity managed by the datastore without being tied to any field values of a JDO instance.

3. Nondurable identity - JDO identity managed by the implementation to guarantee uniqueness in the JVM but not in the datastore.

JDO can be used either in managed or non-managed environments. For example, in order to achieve a benefit from the container-managed transaction and persistence, JDO can be integrated with EJB's in the managed-environment of the J2EE application server. However, this again induces the complexity of EJB.



```
package roster;
public class Player {
  String playerID;
  Address residence;
  // …
}

public class Address {
  String street;
  String postCode;
  // …
}
```

Figure 2.8: Definition of plain `Player` and `Address` classes

As a simple example, Figure 2.8 shows two plain classes `Player` and `Address` that are going to be enhanced to be persistent according to the *plyaddr.jdo* persistence descriptor of Figure 2.9. In this descriptor, the `Player` and the `Address` classes are announced to be persistent. All fields inside these classes are going to be persistent fields.

```
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name=“roster">
    <class name=“Player“ />
    <class name=“Address“ />
  </package>
</jdo>
```

Figure 2.9: Persistence descriptor announces `Player` and `Address` as persistent classes

In the main application of Figure 2.10, the persistence manager object `pm` is being instantiated from a given persistence manger factory `pmf`. A `Player` object `ply` and an `Address` object `addr` are instantiated. Then, the current transaction `t` is retrieved from the persistence manager `pm` and a `begin` demarcation is issued on `t`. The object `ply` is made persistent using `makePersistent` method and according to persistence by reachability, object `addr` is made also persistent. At the end, the transaction is being committed and the persistence manager instance `pm` is closed.

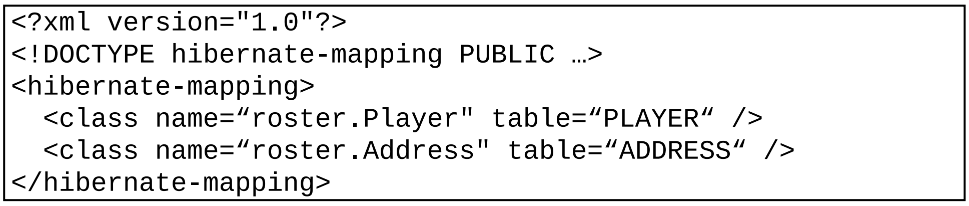
2.4.4.3 Hibernate

Hibernate [Bauer and King (2005); RHM (2007)] is a framework that provides relational persistence for POJO objects. It provides a pool manager that controls the creation and release of persistence sessions. The types that are to be persistent should be defined to extend the `Serializable` interface of the Java API. Those types should contain the setter and getter methods that are going to be used by the session to perform save and retrieve operations on the objects. The programmers have to define their own Hibernate mapping file (cf. Figure

```
package roster;
public class TestPlayer {
  PersistenceManagerFactory pmf;
  PersistenceManager pm;
  Transaction t;
  Player ply;
  Address addr;
  public static void main(String[] args) {
    /*
      some code that initializes the persistence manager
      factory object (pmf)
    */
    pm = pmf.getPersistenceManager();
    try {
      addr = new Address();
      ply = new Player();
      ply.setAddress(addr);
      t = pm.getCurrentTransaction();
      t.begin()
      pm.makePersistent(ply);
      t.commit();
    } finally {
      pm.close();
    }
  }
}
```

Figure 2.10: A test JDO application showing how to persist objects

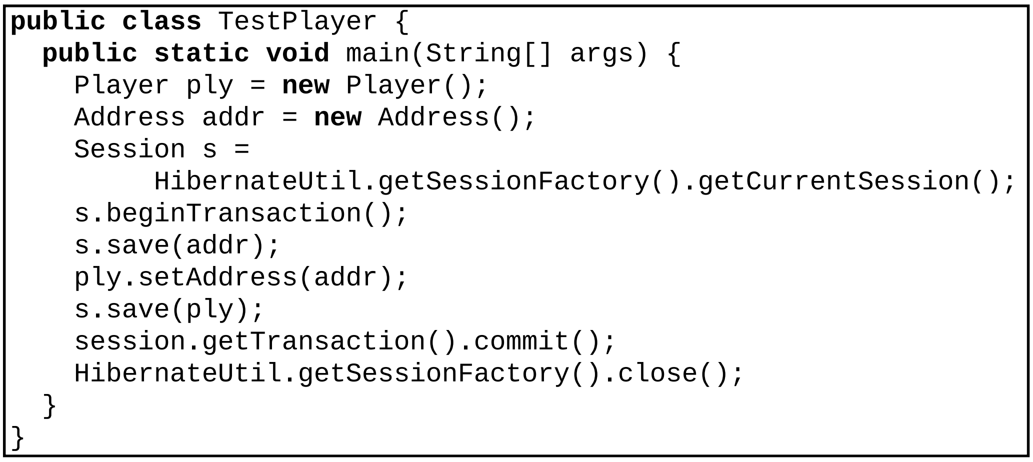
2.11), which maps the persistent classes to the database tables. They are also recommended to follow the naming conventions of setter and getter methods but this is encouraged but not required, since Hibernate can access fields directly.



```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC …>
<hibernate-mapping>
  <class name=“roster.Player" table=“PLAYER“ />
  <class name=“roster.Address" table=“ADDRESS“ />
</hibernate-mapping>
```

Figure 2.11: A definition of a Hibernate class-table mapping

The Hibernate API provides interfaces for persistence and transaction managements. The persistent objects must live inside a specific `Session` object that is instantiated by the programmer. The `Session` interface provides methods `save` and `persist` that can be used to add a given object to a session to make it persistent. A similar method `load` is also provided by the session in order to retrieve a previously saved object from the datastore. Hibernate has its own object-oriented query language (HQL) that must be programmed directly in the base code in a similar way to JDBC statements. All these operations can be run also inside a transaction that is instantiated and demarcated explicitly by the programmer.



```
public class TestPlayer {
  public static void main(String[] args) {
    Player ply = new Player();
    Address addr = new Address();
    Session s =
         HibernateUtil.getSessionFactory().getCurrentSession();
    s.beginTransaction();
    s.save(addr);
    ply.setAddress(addr);
    s.save(ply);
    session.getTransaction().commit();
    HibernateUtil.getSessionFactory().close();
  }
}
```

Figure 2.12: Persisting objects in Hibernate

Figure 2.12 illustrates a simple program example that uses Hibernate to persist objects of `Player` and `Address` classes. It first instantiates a `Session` object. Then, it begins a transaction that surrounds the save operations of the `ply` and `addr` objects before it commits. Finally, the sessions have to be closed before the end of the program by simply closing the session factories.

2.5 Aspect-Oriented Programming for Object Persistence

In the aspect-oriented programming literature, persistence has been identified as a classical candidate for aspectization [Mens et al. (1997); Rashid (2000); Suzuki and Yamamoto (1999)]. The problem of persistence is about where to add the persistence related manipulations to the code. The motivation for aspect-oriented systems to provide better separation of persistence concern is based on the following features of these systems. First, the quantification by means of the pointcuts is used to select the join points inside the base code where persistence manipulations must be involved. Then, the advice is used to specify the persistence concern behavior at the selected join points. Last, the weaving mechanism adds the persistence aspect behavior defined in the advice to the selected join points (cf. Section 2.2).

Following this motivation, a significant effort has been made in the aspect-oriented software development literature to address the problem of object persistence. The proposals for aspect-orientation in object persistence range from solving some concepts of object persistence to frameworks that offer full object persistence to an important extent.

The following are some examples of such proposals. Rashid and Chitchyan (2003) proposed a persistence framework that covers most of the object persistence issues. Soares et al. (2002) proposed an AspectJ implementation of the distribution and the persistence aspects. Some other proposals provide aspectized solutions to the distribution such as [Ceccato and Tonella (2004); Tilevich et al. (2003)]. An aspect-oriented tool for aspectizing distribution was proposed by Benavides et al. (2006). Kienzle and Guerraoui (2002) investigated to what extent aspect-oriented programming using AspectJ can provide a transparent separation of transactions. Last but not least, a domain specific aspect-oriented language called KALA is proposed to aspectize advanced transaction management [Fabry (2005); Fabry and D'Hondt (2006)].

Aspect-orientation has been added to a number of application servers such as JBoss [RHM (2006)] and J2EE [Cohen and Gil (2004)] to support middleware service like persistence in a distributed environment. JAC [Pawlak et al. (2004)] is another aspect-oriented framework that provides the separation of middleware concerns including persistence. Finally, an approach to define aspect-oriented database is proposed in [Rashid (2000, 2004); Rashid and Pulvermueller (2000)].

In the next chapter, three of the above solutions, namely distribution and persistence in AOP (DPA) [Soares et al. (2002)], persistence as an aspect (PAA) [Rashid and Chitchyan (2003)] and Java aspect components (JAC) [Pawlak et al. (2004)], are going to be discussed in detail with respect to the extent to which they support the orthogonal persistence principles. At this point, it is required to introduce these systems. The following subsections give a short introduction to each of these proposals. The related work in Chapter 8 will refer to some different discussions about them, though.

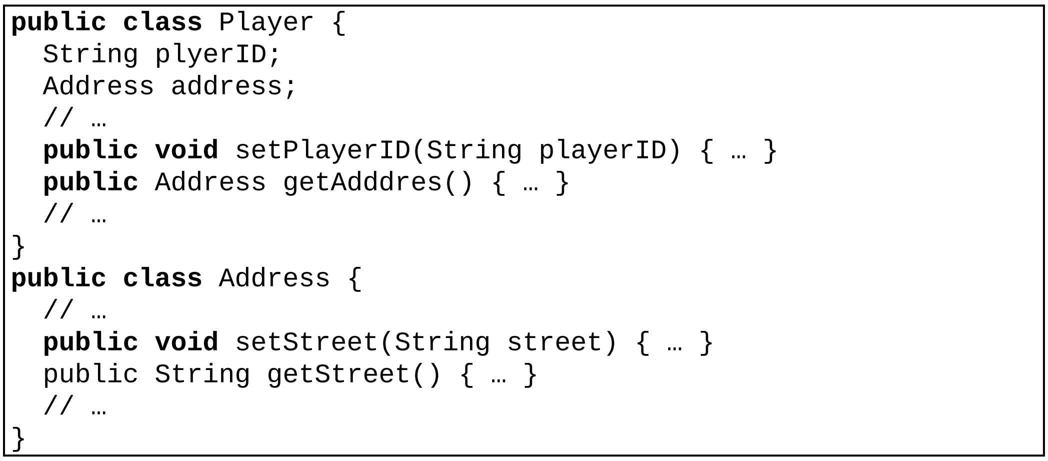
2.5.1 Distribution and Persistence in AOP (DPA)

Soares et al. (2002) implemented an aspect-oriented solution for the persistence and the distribution concerns. The authors used AspectJ to refactor a layered web-based health complaint application. They distinguished between the distribution aspect at the server side and the one at the client side that follows the facade design pattern [Gamma et al. (1994)]. Each aspect offers the remote access service using the Java RMI (Remote Method Invocation) [SUN (1999)].

The distribution aspects consider the object serialization required for the facade parameters. Persistence aspects provide basic persistence functionality: The connection to the used relational database, transaction management, and object state synchronization with the corresponding database. The authors considered the object state synchronization concern as a crosscutting concern between the distribution and the persistence aspects. They chose to solve this dependency as a part of the persistence aspects. Finally, the authors pointed to some weaknesses in the AspectJ language to support good modularization and suggest some proposals, e.g., the need for constructs that add an exception type to a method throws clause.

In [Soares et al. (2002)], the authors issued a number of naming conventions that must be followed. For example, the business layer classes must have the suffix ”`Record`” in order to allow persistence at this layer. Each persistent type must be declared to implement the `PersistentObject` interface in the persistence aspect. The setter and getter methods also must start with ”`set`” and ”`get`”, respectively. Moreover, the developers must also declare that the methods of the facade have serializable parameter and return types inside the server-side persistence aspect by extending their types to implement the `Serializable` interface.

In order to use this framework the developers must extend two abstract aspects: the `AbstractPersistenceControl` and `AbstractTransactionControl` with their own application specific aspects. Moreover, they have to declare the transactional methods that should be run in the context of transactions.

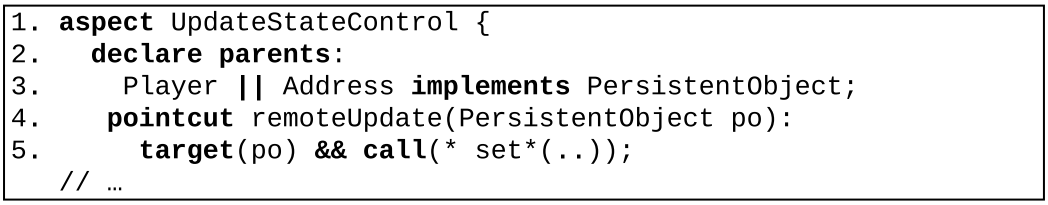


```
public class Player {
  String plyerID;
  Address address;
  // …
  public void setPlayerID(String playerID) { … }
  public Address getAdddres() { … }
  // …
}
public class Address {
  // …
  public void setStreet(String street) { … }
  public String getStreet() { … }
  // …
}
```

Figure 2.13: `Player` and `Address` classes follow the naming convention in DPA

For example, Figure 2.13 shows that the persistent classes `Player` and `Address` are not declared in the base code to be persistent. However, their setter and getter methods should start with `set` and `get`, respectively. The announcement of these types to be persistent is done in the application specific aspect

UpdateStateControl that is shown in Figure 2.14. The declare parents construct makes the Player and Address classes implement the persistent markup PersistentObject. This declaration and the names of the setter methods allow the pointcut remoteUpdate to select all method call join points whose names start with “set” and where target objects are of type PersistentObject.



```
aspect UpdateStateControl {
  declare parents:
    Player || Address implements PersistentObject;
   pointcut remoteUpdate(PersistentObject po):
     target(po) && call(* set*(..));
// …
```

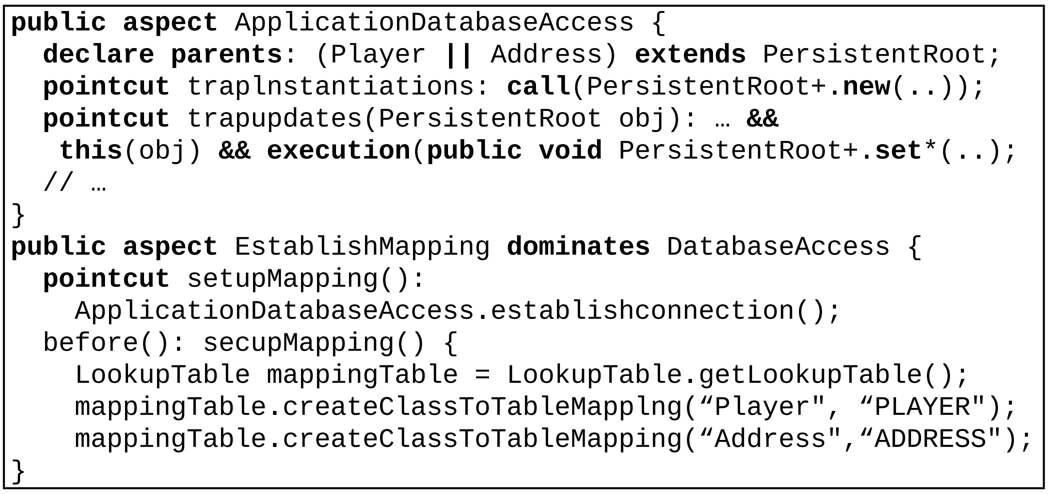
Figure 2.14: The Player and Address classes are made persistent in DPA

For the reasons of simplicity, this example is not concerned with the distribution and transactional aspects, and it considers a simple part of an application that can comply with the structure of the Health Complaint System. In particular, it shows how persistent business logic plain types are declared to be persistent.

2.5.2 Persistence as an Aspect (PAA)

Rashid and Chitchyan (2003) proposed the first attempt for a complete AO persistence framework. This framework is making use of the relational database, and it consists of several aspects that offer different persistence functionalities. These functionalities include database connection, data normalization when mapping object to or retrieving them from tables, persistence operations, SQL translation mechanism, and transaction demarcations. This framework is announced to support persistence by reachability.

As in DPA, the PAA framework relies on the persistent root markup in defining the types that are terms for persistence, precisely, persistent types should be declared to extend the PersistentRoot class. This is done similarly by using the declare parents construct of AspectJ to enumerate all persistent types. PAA also makes use of the naming conventions of the setter and getter methods that must be used inside the base code whenever an object is wanted to be made persistent.



```
public aspect ApplicationDatabaseAccess {
  declare parents: (Player || Address) extends PersistentRoot;
  pointcut traplnstantiations: call(PersistentRoot+.new(..));
  pointcut trapupdates(PersistentRoot obj): … &&
   this(obj) && execution(public void PersistentRoot+.set*(..);
  // …
}
public aspect EstablishMapping dominates DatabaseAccess {
  pointcut setupMapping():
    ApplicationDatabaseAccess.establishconnection();
  before(): secupMapping() {
    LookupTable mappingTable = LookupTable.getLookupTable();
    mappingTable.createClassToTableMapplng(“Player", “PLAYER");
    mappingTable.createClassToTableMapping(“Address",“ADDRESS");
}
```

Figure 2.15: The `Player` and `Address` classes are made persistent in PAA

Figure 2.15 illustrates how to make the types `Player` and `Address` persistent in PAA. The `trapInstantiations` pointcut uses the common persistence marker again, the class `PersistentRoot`, to select all object instantiations of any `PersistentRoot` class. Similarly, the `trapUpdates` pointcut uses this class marker and the naming convention of the setter methods to select the execution join points that update a persistent object. Moreover, the figure shows how persistent types are mapped to corresponding tables.

2.5.3 Java Aspect Components (JAC)

Java Aspect Components (JAC) [Pawlak et al. (2004)] is an aspect-based framework that is used to build distributed Java applications. Distributed objects are also possible to be defined as persistent objects in JAC. The developers use two levels of the JAC framework in order to define distribution and persistence to their POJO objects. On the programming level, the developers define their aspects (aspect components) with pointcuts and wrappers (advice) code. Then these aspects can be refined by configuring them to be applied in a specific scope.

JAC provides a persistence aspect that permits the developer to specify persistent objects. This specification is done by defining a persistent root in a special configuration file. Then the framework applies the persistence by reachability principle to persist all referenced objects from this root. A potential persistent object should be of a type that must comply with some rules, such as having setter and getter methods according to specific naming conventions, i.e., the names of these methods should start with “set” and “get”, respectively.

JAC also permits to define a single class or a package to be persistence. Moreover, it allows the developers to make specific objects persistent. This is done by specifying the name of the object that starts with the name of the class followed by the special character ”#” and the number that represents the sequence of the instantiation of the object. For example, the first instantiated object from of type Player is named in JAC as player#0, the second is player#1 and so on.

```
// declare Player as persistent class
1. MakePersistent roster.Player;
// define the aspect
2. public class RosterAC extends AspectComponent {
3.  public RosterAC() {
4.   pointcut(“.*", // all objects
5.            “Player", // classe Player
6.            “set*(*):void", // setter methods
7.            “ObjChanged", // the name of wrapper (advice)
8.            null, // no Exception handler
9.            false); // wrapper cardinality, here is singleton
10.  }
11. class ObjChanged extends Wrapper { // advice code
12.  public Object invoke(MethodInvocation mi) throws Throwable
13.  { // do something... }
14. }
15.}
```

Figure 2.16: Persisting Player objects in JAC

In Figure 2.16, the JAC configuration file specifies that the Player is to be persistent. In the aspect component RosterAC, the given pointcut (line 4) selects all method invocations (line 12) join points of any object (line 4) of type Player (line 5) where these methods starts with set (line 6).

2.6 Chapter Summary

This chapter introduced many concepts that are related to the persistence problem domain. It first defined the term separation of concerns and showed its importance to the field of software engineering in Section 2.1. Then, Section 2.2 described the meaning of aspect-oriented software development (AOSD) and a number of its technologies. It discussed also how AOSD technologies can be used to support the separation of concerns in many fields of computer science.

The focus also lay on one of these technologies: the aspect-oriented programming (AOP) in Section 2.2.1, which is going to be the main subject of the thesis. Then, in Section 2.2.2, AspectJ programming language was introduced as an example of AOP systems.

In Section 8.1, object persistence was introduced along with its different related issues as a crosscutting concern that must be handled separately from the object logic of the applications. A number of current object-oriented approaches to solve object persistence have been discussed in Section 2.4 with some emphasis on current widely-used persistence frameworks: EJB, JDO and Hibernate in Section 2.4.4.

Finally, Section 2.5 gave an overview of the current solutions for object persistence using the available AOP systems.

In the next chapter, a more concrete discussion on the aforementioned techniques is going to be presented to show the extent to which these techniques comply with the principles of orthogonal persistence.

Chapter 3

Problem Description

In this thesis, the first premise of orthogonal object persistence is that the persistence system must comply with the orthogonality principles: Type orthogonality, persistence independence and transitivity [Atkinson (2000); Atkinson and Morrison (1995)]. From an aspect-oriented programming perspective, this requirement meets with the obliviousness property [Filman and Friedman (2000)]: by examining the base code, one cannot tell that the persistence aspects are executed at specific points in the code. Precisely, the base code does not need to be prepared in order for its objects to be made persistent.

This chapter presents a number of examples that show how current aspect-oriented persistence systems fail to fulfill the obliviousness characteristic of the aspect-oriented programming, in other words, these systems defeat the orthogonal persistence principles [Al-Mansari et al. (2007a)]. Hence, two more examples from the object persistence domain related to the concurrency control issue are presented to justify the need for abstraction over the non-local join point properties that are based on the object relationships [Al-Mansari et al. (2007b)].

This chapter is organized as follows. Section 3.1 describes the object model that is going to be used in the examples throughout the rest of the thesis.

Section 3.2, discusses the orthogonality between the base code and the current conventional and aspect-based persistence frameworks. In Section 3.2.1, these systems are examined to show the extent to which the base code needs to be prepared for persistence at the type-level. Section 3.2.2 examines this property at the code-level.

Section 3.3 explores the importance of the object relationships for object persistence. This is discussed with respect to two problem domains: aspectizing pure persistence issue, i.e. CRUD operations in Section 3.3.1 and aspectizing the persistence related issue of concurrency control in Sections 3.3.2 and 3.3.3.

Section 3.4 argues about the importance of object information in the aspectization of other crosscutting concerns.

Section 3.5 states the problem concisely. Finally, the chapter is summarized in Section 3.6.

3.1 The Object Model of the Motivating Examples

In the rest of the thesis, the illustrative examples are making use of the class diagram in Figure 3.1. The class hierarchy in the figure is adopted from Kifer et al. (1992) and represents a company object model.

Each `Company` object has a `headquarter` address, a `president` of type `Employee` and a number of `divisions`. Each `Division` object is associated with a `location` address, a `manager` of type `Employee`, a number of `employees` and `customers`. A `Customer` object has two address fields `billTo` and `shipTo`. `Customer` and `Employee` are subtypes of `Person` that is associated with a `residence` address and a number of `ownedVehicles`. The `Person` class also has a many-to-many relationship called `familyMembers` with itself. There are two types of vehicles, `Automobile` and `Motorbike`.

Most of the relationships between the objects in this model are associations, i.e. one-directional relationships. It must be noted that one consequence of this *ownership relationship* is that the owned objects are not aware of their owner objects. This means that the relationship information, i.e. the reference to the owned object and the name of the relation, is part of the owner object. For example, a given `Address` object does not know whether it is a part of a `Person` object, a `Division` object or a `Company` object.

The thesis is also making use of the terms *direct relationships* and *indirect relationships* that need to be defined (cf., e.g., [Stein et al. (2006)]). The direct

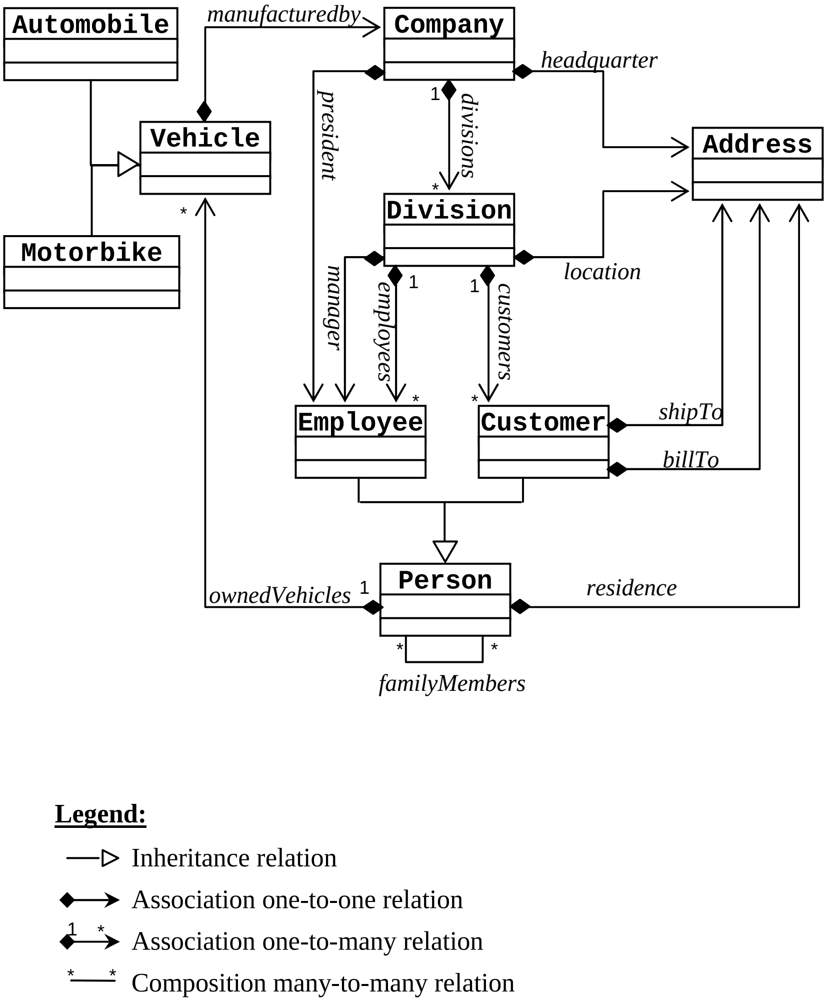


Figure 3.1: Problem domain class diagram

relationship between objects a and b means that the object b represents the value of a corresponding field of the object a. For example, the relationship between the Company object com and its president Employee object in the object graph of Figure 3.2 is a direct relationship. On the other hand, the indirect relationship between two objects a and b means that there is at least one inner object x, where x represents the value of a field in object a and b represents the value of a field in x. For example, in the object graph of Figure 3.2, the relationship between the Company object com and the address addr is an indirect relationship where pres is an inner object in this relation. Indirect relationships are defined by the transitive closure relation.

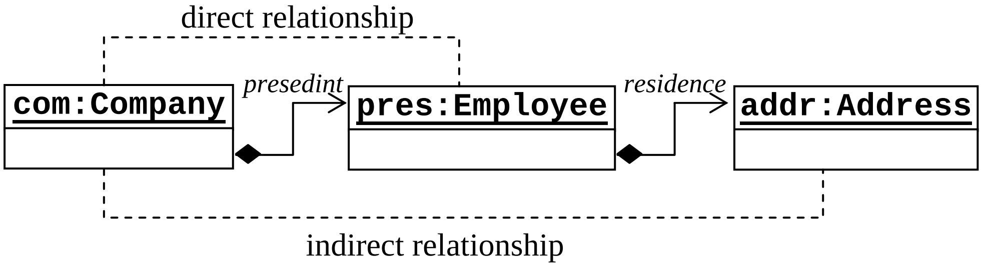


Figure 3.2: Direct and indirect relationships between objects in object graph

Finally, the object model contains cycles and duplicate associations between the object, which covers all possible situations when dealing with object relationships in different examples throughout the thesis. For example, in Figure 3.3, there is a cycle that runs through the objects com, div1, cust1, car1 then com again. Moreover, according to the figure, there are two associations from the object cust1 to the same object addr, namely shipTo and billTo.

The above object model (cf. Figure 3.1) is going to be used throughout the rest of this chapter in order to illustrate the problems from which current aspect-based solutions suffer when trying to fulfill the principles of orthogonal persistence. It must be noted that in most of the used examples, AspectJ code is being used for the purpose of understandability, since AspectJ is known as the most dominant aspect-oriented programming language at the time being. For the purpose of comprehension, examples from some current object-oriented persistence mechanisms are presented when required.

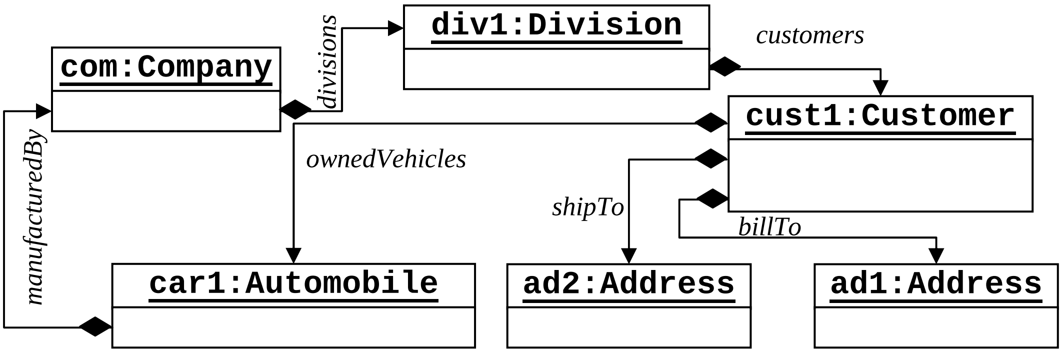


Figure 3.3: A cycle and a duplicate relationship in an object graph

3.2 Preparing Objects for Persistence

The obliviousness property [Filman and Friedman (2000)] of aspect-oriented programming means that the base code does not contain any indications that it is going to be affected by any aspect. With respect to persistence aspects, this property seems to promise the orthogonality between the base code and its aspectized persistence manipulation, i.e. orthogonal persistence. Accordingly, there are a number of aspect-based attempts to provide orthogonal object persistence (cf. Section 2.5).

This thesis will investigate whether these solutions comply with the obliviousness property, i.e., whether the base code is oblivious to the persistence aspects. In other words, whether the base code does not need to be prepared in order to receive persistence functionality from the persistence aspects of these systems.

The thesis distinguishes between the following two different ways of how applications are required to be prepared in order to provide the persistence functionality:

The type-level preparation. It is about introducing persistence-related parts to the types' definitions. For example, declaring a persistent type to implement persistence-specific interfaces. The thesis divides the types into two categories, developer-defined types that are defined by the developer of the persistence application himself and third-party types that are imported from other packages and libraries, e.g, a tree library. The term third-party code is used here since the thesis considers two different related parties, namely the persistence layer framework or tool and the persistent application.

The code-level preparation. It is concerned with the inclusion of some code that participates in deciding when the objects should be made persistent. This kind of preparation can be seen statically by examining the base code. For example, the base code contains an invocation to a method that makes a given object persistent. The code is also divided into two similar categories of types, i.e., the developer-code and the third-party code.

The following sections discuss these different kinds of preparations in more detail and assess whether the current approaches for object persistence do fulfill the requirements of orthogonal persistence that ensures the orthogonality between the business logic objects and their persistence.

This assessment considers a number of object-oriented and aspect-oriented persistence systems. The object-oriented systems are EJB,JDO, and Hibernate. The aspect-oriented systems are the PAA persistence framework, the persistence aspects in DPA , and the JAC framework.

Sections 2.4 and 2.5 introduced these systems, respectively. The discussion of orthogonal persistence support by object-oriented persistence solutions is provided for reasons of comprehension, though, the discussion will not consider their problems as a part of the thesis' main motivation.

3.2.1 Preparing Objects at Type-Level

One of the main problems of the conventional persistence systems such as EJB, JDO, and Hibernate is that they require the developers to define explicitly the types and classes whose objects are to be made persistent. This is somehow similar to the case of aspect-oriented systems. The following subsections illustrate these limitations.

3.2.1.1 EJB Type-Level Preparation

In EJB, the developers have to follow certain rules in order to prepare a bean for persistence as mentioned in the last chapter. Figure 3.4 shows an example of how to declare the PersonBean to be persistent.

```
public abstract class PersonBean implements EntityBean {
  public abstract Address getResidence();
  public abstract void setResidence(Address residence);
  // …
  public abstract Collection ejbSelectVehicle(LocalPerson person)
      throws FinderException;
  // …
} // PersonBean class
public interface LocalPersonHome extends EJBLocalHome {
  public LocalPerson findByResidence(Address residence)
      throws FinderException;
  // …
} // LocalPersonHome interface
public interface LocalPerson extends EJBLocalObject {
  public Address getResidence();
  // …
} // LocalPerson interface
```

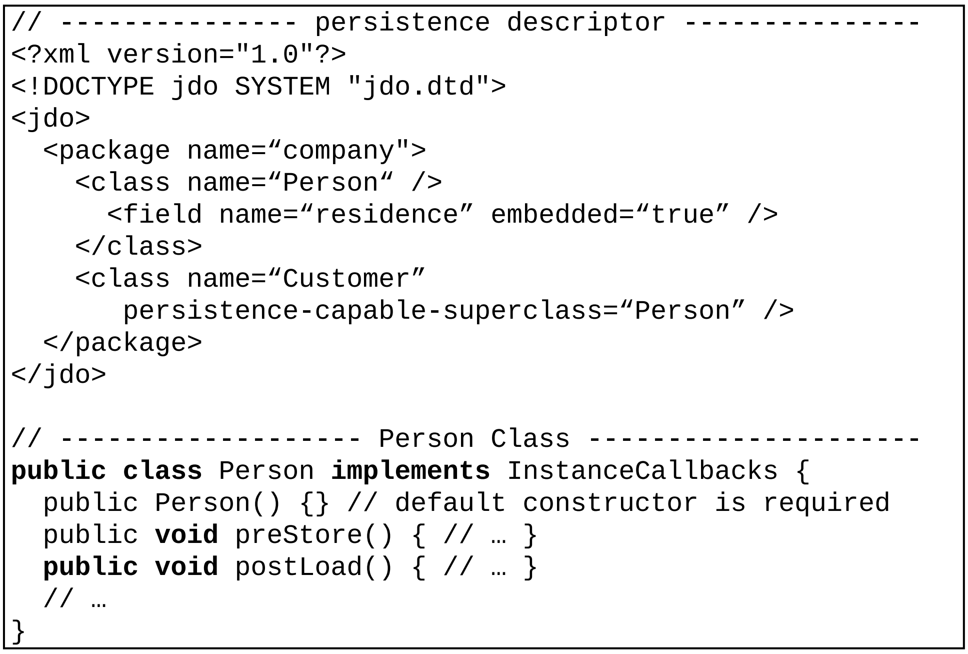
Figure 3.4: Preparing objects for persistence at type-level in EJB

In such systems, the persistent types should be prepared to identify objects that should be made persistent, or to identify methods that can have an effect on the persistent objects. Figure 3.4 illustrates that the PersonBean implements the EntityBean interface and its accessor, selector, and finder methods follow the EJB naming conventions. Moreover, it is required to define the local and home interfaces for the PersonBean.

Note that the thesis considers mainly version 2.0 of EJB. Nevertheless, it is worth to mention that EJB 3.0 [SUN (2006a)] simplifies the process of defining persistent types by means of annotating persistent types. For example, the entity beans must be annotated with @Entity. Such annotations are also required for specific field and method declarations in persistent types, e.g. @Id to declare a given field as a primary of the entity. These annotations are used by the container to propagate suitable persistence contexts across the bean. Moreover, the persistent types must maintain the naming conventions of the getter and setter methods of the persistent types. All of the above is still considered as a type preparation for persistence, since the developers need to update the persistent types explicitly.

3.2.1.2 JDO Type-Level Preparation

As mentioned earlier, JDO promises transparent persistence for plain Java objects (cf. Section 2.4.4.2). Nevertheless, the developers still have to declare the persistence objects at the type level even though in an easier way than the one of EJB. The developers can define a persistent type either by directly declaring it as an implementor of the `PersistenceCapable` interface or by declaring this type as persistent in the persistence descriptor. This is considered as a preparation for persistence at type-level.



```
// --------------- persistence descriptor ---------------
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name=“company">
    <class name=“Person“ />
      <field name=“residence” embedded=“true” />
    </class>
    <class name=“Customer”
       persistence-capable-superclass=“Person” />
  </package>
</jdo>

// ------------------ Person Class --------------------
public class Person implements InstanceCallbacks {
  public Person() {} // default constructor is required
  public void preStore() { // … }
  public void postLoad() { // … }
  // …
}
```

Figure 3.5: Preparing Objects for persistence at type-level in JDO

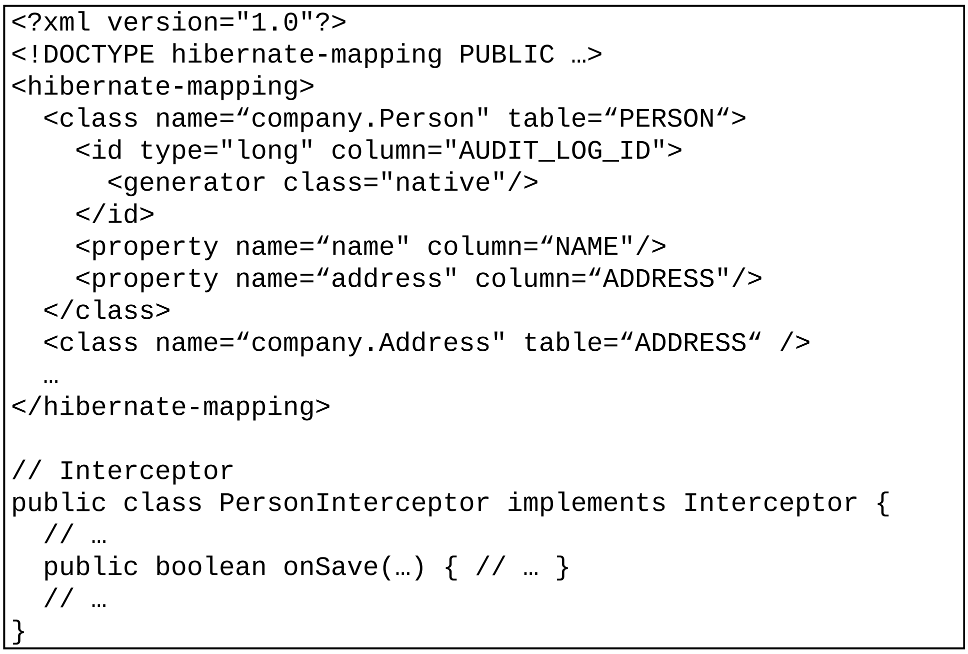
Figure 3.5 illustrates how the objects of the `Person`, `Address`, and `Customer` types are prepared at type-level using the persistence descriptor. Notice that the descriptor also specifies how the persistent objects of these types are related. The composition relation between `Person` objects and their `residence` fields that are of type `Address` is specified by the attribute "`embedded`". JDO persistence manager will treat the residence objects of each persistent `Person` as *second-order class* persistent objects, i.e. `Address` object should not be persistent unless it is referenced by a persistent owner `Person` object.

Persistence capable classes in JDO usually must contain a default constructor (no-argument constructor) to be used by the JDO implementation. For example, if a default constructor is not available, the enhancer will add it with a default call to the `super()` method. In the later case, the developer must add a default constructor to the non-persistence capable direct superclass of the persistence capable class.

In some situations, it is required that objects are made aware of specific lifecycle events occurring to them, so that the developer can invoke an appropriate action [Roos (2002)]. For example, validating the object state before the transaction is committed. Accordingly, persistent types must implement the `InstanceCallbacks` interface as shown in Figure 3.5. Therefore, the developers have to implement this interface, which is considered a type level preparation.

3.2.1.3 Hibernate Type-Level Preparation

The situation in Hibernate is almost the same as shown in JDO. The application developers define the persistent types by using metadata as depicted in Figure 3.6. In general, the persistent types in Hibernate are used in `HttpSession` or passed by value using RMI in distributed environments such as web-based applications. As a consequence, the persistent type must be modified to implement the `Serializable` interface. Also, it is recommended to use property accessor's methods of an EJB style, i.e. `setProperty` and `getProperty`. These naming convention guidelines allow generic tools like Hibernate to easily discover and manipulate the property value. Moreover, the persistence class in Hibernate must have a default constructor.



```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC …>
<hibernate-mapping>
  <class name=“company.Person" table=“PERSON“>
    <id type="long" column="AUDIT_LOG_ID">
      <generator class="native"/>
    </id>
    <property name=“name" column=“NAME"/>
    <property name=“address" column=“ADDRESS"/>
  </class>
  <class name=“company.Address" table=“ADDRESS“ />
  …
</hibernate-mapping>

// Interceptor
public class PersonInterceptor implements Interceptor {
  // …
  public boolean onSave(…) { // … }
  // …
}
```

Figure 3.6: Preparing objects for persistence at type-level in Hibernate

As in JDO, it is necessary in many situations to have a knowledge about the current states of the persistent objects that is delimited by the so co-called *Callback* methods like `onSave`. The developers either have to prepare persistent classes to implement interfaces such as `Lifecycle` and `Validatable`, or to add separate classes that implement the `Interceptor` interface as shown in Figure 3.6. This is a sort of type prepartion for persistence, since the developers of the persistent types have to implement these special interfaces.

3.2.1.4 Summary of OO-Systems Type-Level Preparation

In the case of object-oriented persistence solutions, it is noticeable that these solutions do not comply with the orthogonal persistence requirements due to the following:

1. These solutions break the principle of type orthogonality because only objects of types that follow the restrictions made by these persistence systems can be persisted. The developers have to prepare both their own

defined types and the imported third-party types. This preparation includes implementing certain persistence system specific interfaces such as `EntityBean` in EJB, `InstanceCallbacks` in JDO, and `Serializable` in Hibernate. Preparing types for persistence includes also to follow certain naming conventions for accessor methods in EJB and Hibernate. Moreover, the persistent types must include a default constructor like in JDO and Hibernate.

2. These systems also break the persistence independence principle for several reasons. First, the persistent types cannot be used in a non-persistence environment, as it is the case with the `PersonBean` in EJB. Second, the developers must have knowledge of the structure of the imported third-party types, which in turn does not promote code reusability. Third, the developers have to be aware of internal aspects of the persistence framework, e.g. the lifecycle state of persistent objects in JDO and Hibernate. Last but not least, in many situations, even the non-persistent type that is a direct supertype of a persistent type must be prepared to fulfill the persistence system requirements.

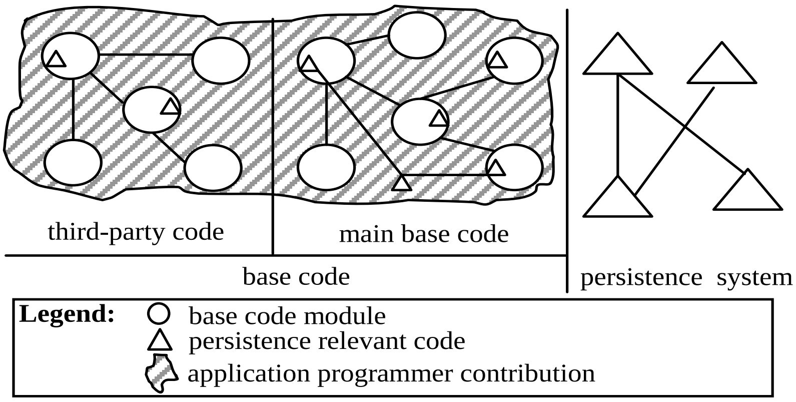


Figure 3.7: Preparing objects for persistence at type-level in OOP frameworks

Figure 3.7 sketches this situation. The persistence application is divided into two layers: The left one represents the application base code modules (the circles) and the right layer represents the persistence framework modules (the triangles). The application base code modules in turn are divided into two different categories: The left represents the imported third-party code and the right one represents the code that is written by the application developer. The shaded area represents the application developer's contribution in defining persistent types of her/his application.

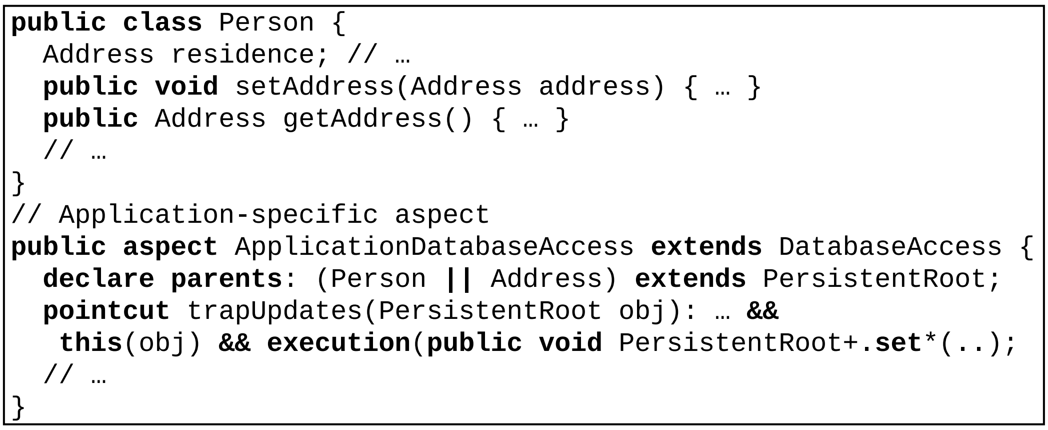
As mentioned above, the developer has to add explicit persistence system related modifications (the small triangles) to the base code, e.g. extending the `EntityBean` interface. This extension may affect more than one module in the base code, which is indicated by multiple occurrences of the small triangles inside the base code modules. Such preparation affects the two different categories of the application base code. The occurrence of the small triangles in the base code reflects the non-orthogonality between the base code and the persistence system.

3.2.1.5 Type-Level Preparation in PAA

In PAA [Rashid and Chitchyan (2003)], the developers should designate the persistent classes by using the `declare parents` construct to let them extend the `PersistentRoot` class. This kind of enumeration represents a type-level preparation for persistence. There is also another minor problem relating to Java in this framework, i.e. the developers have to be sure that they can introduce a new superclass to the target classes, these classes must be inherited directly from the general superclass `Object` so that `declare parents` can have the desired affect.

Moreover, the developers have to apply the naming guidelines to the accessor methods of the persistence types in a similar way, e.g. to EJB. These methods should start with corresponding "`set`" and "`get`" prefixes.

Figure 3.8 illustrates these points with respect to the `Person` and `Address` persistent types. Note that the accessor methods naming conventions are enforced by the `trapUpdates` pointcut specification that is making use of the property based crosscutting, i.e. the name of the accessor method.



```
public class Person {
  Address residence; // …
  public void setAddress(Address address) { … }
  public Address getAddress() { … }
  // …
}
// Application-specific aspect
public aspect ApplicationDatabaseAccess extends DatabaseAccess {
  declare parents: (Person || Address) extends PersistentRoot;
  pointcut trapUpdates(PersistentRoot obj): … &&
   this(obj) && execution(public void PersistentRoot+.set*(..);
  // …
}
```

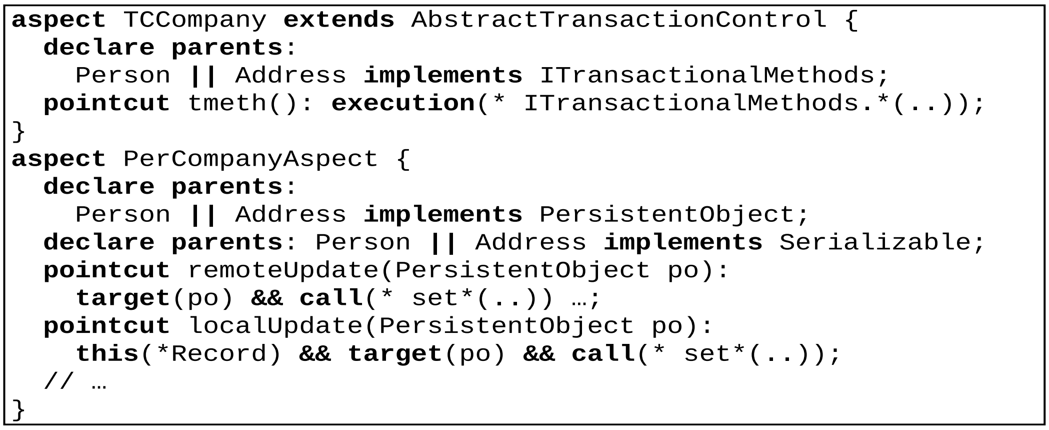
Figure 3.8: Preparing objects for persistence at type-level in PAA

3.2.1.6 Type-Level Preparation in DPA

The developers in DPA [Soares et al. (2002)] have to enumerate the persistent types in an application-specific aspect in order to induce them to implement the PersistentObject interface. In addition to that, the persistent objects must be prepared to be serialized by making their types implement the Serializable interface. This is because these objects in this system are terms to be used within RMI methods. Moreover, persistent types must also implement the ITransactionalMethods interface in order to designate the transactional methods.

There are also certain guidelines for naming some types and the accessor methods of the persistent types. The business layer classes must have names that have the postfix Record in order to be used to persist the persistent objects in the context of these business classes. This means that the developer must also be sure that every imported third-party type follows this naming convention, otherwise there might be updates on persistent objects in the business layer classes so that these updates will not be made persistent. Other naming conventions restrict the setter and getter methods to start with set and get, respectively, when defining the interface of a type whose objects are to be made persistent.

All of these preparations are illustrated in the example of Figure 3.9. The example shows how to prepare the Person and the Address types for persistence in DPA.



```
aspect TCCompany extends AbstractTransactionControl {
  declare parents:
    Person || Address implements ITransactionalMethods;
  pointcut tmeth(): execution(* ITransactionalMethods.*(..));
}
aspect PerCompanyAspect {
  declare parents:
    Person || Address implements PersistentObject;
  declare parents: Person || Address implements Serializable;
  pointcut remoteUpdate(PersistentObject po):
    target(po) && call(* set*(..)) …;
  pointcut localUpdate(PersistentObject po):
    this(*Record) && target(po) && call(* set*(..));
  // …
}
```

Figure 3.9: Preparing objects for persistence at type-level in DPA

3.2.1.7 Type-Level Preparation in JAC

Similarly, in JAC, the programmers must enumerate the persistent types in the configuration file as shown in the first and the second code lines of Figure 3.10. The programmers are also encouraged to follow the naming conventions for defining the fields' access methods of the persistent types, otherwise they have to define such naming conventions in the configuration files of the framework which is still a type preparation.

Note that in the aspect specification, e.g. `CompanyAC` in Figure 3.10, the programmers are forced to dedicate separate pointcuts to each persistent type because persistent types do not have a common persistent root such as the one used in the previous two proposals.

3.2.1.8 Summary of AO-Systems Type-Level Preparation

As illustrated in the last three subsections, the persistence application developers in the aspect-oriented frameworks still have to consider preparing the persistent types. Despite of the fact that this preparation is minimized with respect to the base code, these systems shift the preparation task to the persistence layer: Instead of having an explicit declaration for each persistent type within its code,

```
//Declare Person and Address as persistent in configuration file
MakePersistent company.Person;
MakePersistent company.Address;

public class CompanyAC extends AspectComponent {
  public CompanyAC() {
    pointcut(“.*", “Person", “set*(*):void",
             “ObjChanged", null, false);
    pointcut(“.*", “Address", “set*(*):void",
             “ObjChanged", null, false);
  }
  class ObjChanged extends Wrapper {
    public Object invoke(MethodInvocation mi) throws Throwable
    { // do something to persist the update ... }
  }
}
```

Figure 3.10: Preparing objects for persistence at type-level in JAC

the enumeration of such types becomes part of those aspects that introduce the corresponding type hierarchy.

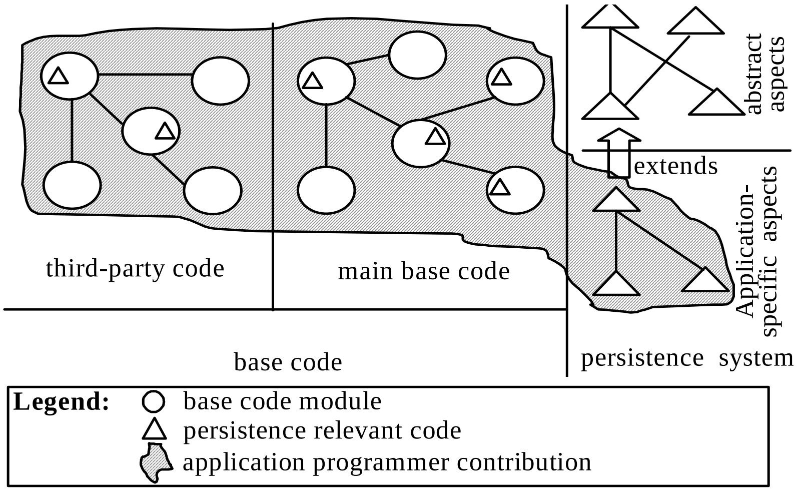


Figure 3.11: Preparation of persistent objects at type-level in AOP frameworks

This phenomenon is depicted in Figure 3.11. The application developer’s participation (the shaded area) includes the application-specific extension of the persistence aspects in the persistence layer. This means that the developer has to extend the abstract aspects of the persistence framework in order to prepare the types for persistence. Besides, the developer still has to add minor modifications to the base code persistent types, e.g. the developer must ensure that the setter and getter methods follow the naming conventions as it is required by the used aspect-oriented persistence frameworks.

From Figure 3.11, one can conclude that the current aspect-oriented persistence frameworks do not follow the principle of orthogonal persistence. This is due to the following:

1. They break the principle of type orthogonality. The reason is that persistent types still have to be specified by enumerating them explicitly in the aspects.[1] Moreover, persistent types have to be modified in many cases either directly inside the base code layer, e.g. to follow the naming conventions of accessor methods in the three systems or of the business layer classes in DPA. Any other types that do not fulfill these requirements cannot be made persistent, hence, breaking the principle of type orthogonality.

2. These systems compromise the persistence independence principle too. The rationale behind this is that following the naming conventions of defining the accessor methods in the three frameworks and the business layer objects in DPA requires that the application developer has to modify the base code type that tend to be persistent. These types include the developer-defined as well as the third-party types. This in turn breaks the reusability of those imported types, hence, breaking the persistence independence principle.

3.2.2 Preparing Objects at Code-Level

This section explores the extent to which current object-oriented and aspect-oriented persistence frameworks comply with the orthogonal persistence principles with respect to preparing objects for persistence at the code level. As in the previous section, the discussion considers the object-oriented frameworks first followed by the aspect-based systems.

[1] This is a result of the so-called *enumeration-based crosscutting* [Gybels and Brichau (2003)].

3.2.2.1 EJB Code-Level Preparation

The best case of separating persistence in EJB is to use the container-managed persistence. Even in this case, the base code must include explicit invocations to the setter and the getter methods instead of accessing the object fields directly in order to persist the corresponding objects. Moreover, the developers have to be sure that this constraint is fulfilled inside the class themselves, i.e., any access to a field inside its class should be done by using the setter and the getter methods. The instantiation of a persistent bean must be done by using the remote method `create()`, which in turn invokes the `ejbCreate` method inherited from the `EntityBean` interface. Similarly, the relationship fields of the entity beans can not be accessed directly in the base code, rather, the setter and getter methods must be used.

There are several kinds of code preparation in EJB. For example, in bean-managed persistence, developers have to add the persistence operations explicitly, e.g. to create a connection to the database or to demarcate transactions. Moreover, in order for the remote clients (e.g. other computers or different processes in the same computer) to work with EJB's, the developers have to apply the Java Naming and Directory Interface (JNDI) [SUN (2006a)] service that is used to locate resources, such as remote objects, on networks. This is applied to container and to bean-managed persistence, and it is considered also as a preparation the code for EJB persistence framework. All these preparations couple the base code and the EJB persistence framework.

Figure 3.12 illustrates a simple example of creating a container-managed persistence person bean. The first five lines in the `try` block show how to create a single object of the `Person` bean. The rest of the block shows that the setter methods are used in setting the name and the address properties of that person and the finder method is used to retrieve the person from the database.

It must be mentioned that EJB 3.0 [SUN (2006a)] also uses the conventional setter, getter, selector, and finder methods in the base code. Moreover, persistence operations such as update and retrieval operations as well as transaction demarcations should be invoked on an instance of `EntityManager` inside the code. Instances of `EntityManager` are created from an `EntityManagerFactory` object.

```
public class PersonClient {
  public static void main(String [] args) {
    try {
      Context jndiContext = getInitialContext();
      Object ref = jndiContext.lookup(“PersonHome");
      PersonHomeRemote home = (PersonHomeRemote)
        PortableRemoteObject.narrow(ref, PersonHomeRemote.class);
      PersonRemote p1 = home.create(new Integer(1));
      p1.setName(“Green Master");
      p1.setAddress(new Address());
      Integer pk = new Integer(1);
      PersonRemote p2 = home.findByPrimaryKey(pk);
    } catch (Exception e) { re.printStackTrace(); }
  }
}
```

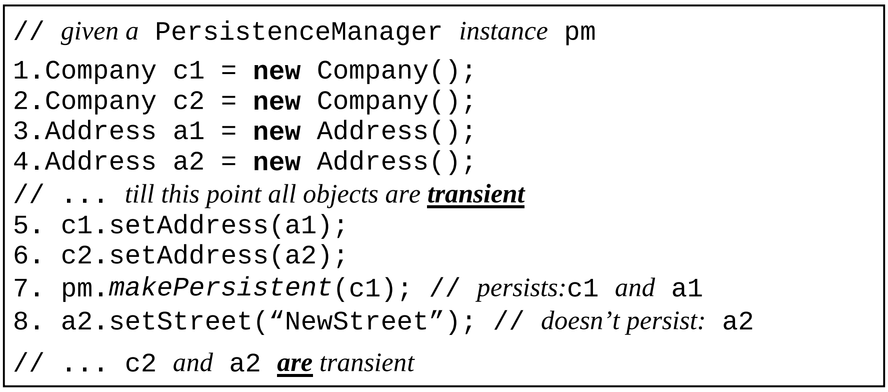
Figure 3.12: Persisting objects in EJB explicitly at code-level

3.2.2.2 JDO Code-Level Preparation

In conventional persistence systems such as JDO [Jordan and Russell (2003)], the developers explicitly invoke methods that persist instances of persistent types. For example, by invoking the method `makePersistent` of a persistence manager in JDO with the object as an argument. All other persistence operations must be explicitly added to the code, e.g. those used for object retrieval and query. These operations also include the lifecycle state related methods inherited from the `InstanceCallbacks` interface.

The persistent base application code in JDO is also prepared for persistence by using explicit persistence-related instances and commands. For example, the persistent objects must be manipulated inside the context of `PersistenceManager` instances, which in turn must be part of a `PersistenceManagerFactory` instance. Also, the transaction manipulations are specified explicitly by the developer.

The JDO community argues that the persistence capable main business classes contain no such signs of the presence of the JDO framework. Nevertheless, the thesis also considers the contribution of the persistent application developer who is still responsible for defining persistence-related manipulations inside a part of the base application code. This contribution covers the code developed by the developer as well as the imported third-party code.



```
// given a PersistenceManager instance pm
Company c1 = new Company();
Company c2 = new Company();
Address a1 = new Address();
Address a2 = new Address();
// ... till this point all objects are transient
c1.setAddress(a1);
c2.setAddress(a2);
pm.makePersistent(c1); // persists:c1 and a1
a2.setStreet("NewStreet"); // doesn't persist: a2
// ... c2 and a2 are transient
```

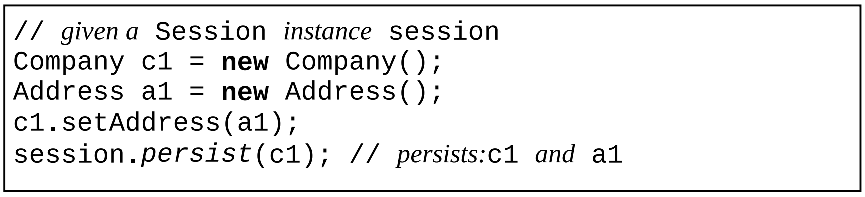
Figure 3.13: Persisting objects in JDO explicitly at code-level

Figure 3.13 shows an example where company objects and address objects are instantiated. Since the company instance `c1` is made persistent (by passing it as a parameter to `makePersistent`), its referenced address object `a1` is persistent as well by means of persistence by reachability. Since the company instance `c2` and the address `a2` are not explicitly requested to be made persistent, they remain transient. Therefore, it is easy to figure out the presence of persistence framework by looking at the code.

3.2.2.3 Hibernate Code-Level Preparation

The situation in Hibernate is similar to JDO. In order to persist a given object, the developer invokes the method `save()` or `persist()` on a Hibernate `Session` object with the persistent object as an argument. Moreover, the persistent objects in Hibernate must also be manipulated in the context of special persistence-related objects, e.g. `Session`, `SessionFactory` and `Transaction`.

Figure 3.14 illustrates another example of how to specify the persistent objects at the code-level in Hibernate. In this case, the developer, again, explicitly invokes the method `persist` on a given session with the object `c1` as an argument. In both examples, the object `c1` as well as its closure will be persistent, which can be figured out by looking at the code.



```
// given a Session instance session
Company c1 = new Company();
Address a1 = new Address();
c1.setAddress(a1);
session.persist(c1); // persists:c1 and a1
```

Figure 3.14: Persisting objects in Hibernate explicitly at code-level

3.2.2.4 Summary of Conventional Systems Code-Level Preparation

The main problem with conventional object-oriented persistence frameworks is that they require the code to contain specific signs when to persist objects, hence, the orthogonality between the base code and its persistence is not fulfilled, which means that the code will not be reusable. This breaks the principle of persistence independence. Therefore, these systems do not comply to the orthogonal persistence requirements.

Defeating persistence independence in conventional persistence frameworks has other reasons. For instance, the developers have to know the internal structure and characteristics of the object-oriented persistence systems. For example, the developers must be aware of the lifecycle states of the persistent objects that are manipulated by the `InstanceCallsback` methods.

Figure 3.15 illustrates this situation. The base code in the left side of the figure contains the preparation for persistence. Note that this figure is the same as the type-level preparation and the difference is that the code preparation does not change the type structure of the base code. This is reflected from the figure where the persistence-related code (triangles) is part of the available code modules.

3.2.2.5 Code-Level Preparation in PAA

Similarly, in PAA [Rashid and Chitchyan (2003)], the base code must include explicit invocations to the setter and the getter methods instead of accessing the fields directly in order to persist the corresponding objects. As in object-oriented systems, the developers have to be sure that the access to the persistence fields inside their classes must be done by using the setter and getter methods.

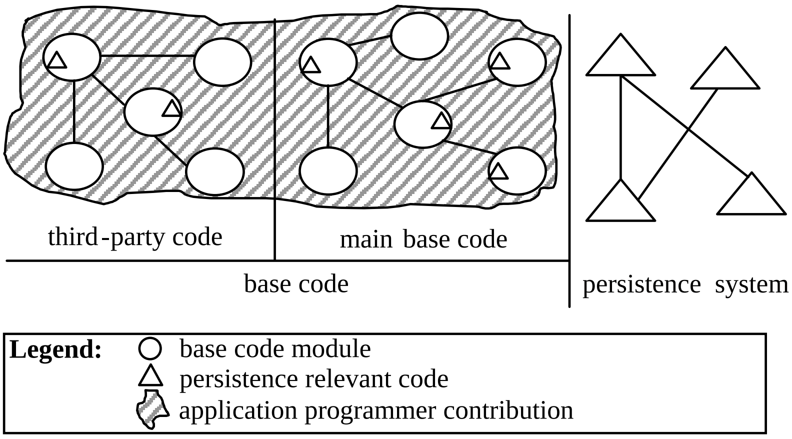
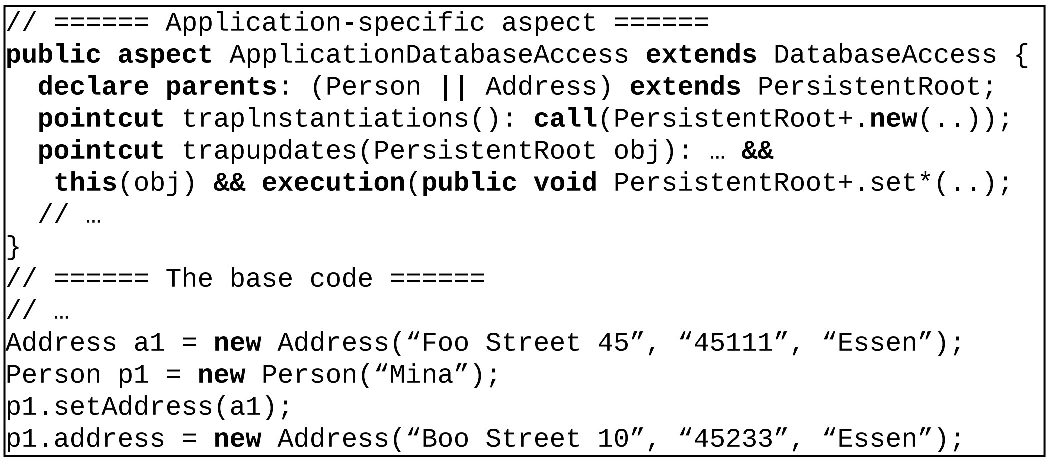


Figure 3.15: Preparing objects for persistence at code-level in OOP frameworks

Accordingly, the developers have to check all accesses to the persistence fields in the base code to be sure that these accesses meet the framework requirement.

For example, consider the aspect code fragment of Figure 3.16, which is taken from Rashid and Chitchyan (2003). The classes `Person` and `Address` are declared to extend the persistence root class `PersistentRoot`. The `trapInstantiations` pointcut selects all instantiation join points of `PersistentRoot` objects in order for the associated advice to persist the instantiated objects. As explained before, the `trapUpdates` pointcut selects all method call join points to the setter methods of the persistent classes so that the associated advice persists these changed objects.

The first two sentences in the base code will be selected by the pointcut `trapInstantiations` and both `a1` and `p1` become persistent objects. Likewise, the third statement in the base code will be selected by the `trapUpdates` pointcut and the set operation to the `address` field of the person object `p1` will be made persistent. This is not the case for the last statement, where the field `address` of the persistent object `p1` is directly accessed. This change is made to a persistent object, however, this change will not be selected by the persistence aspect `trapUpdates` pointcut. In order to overcome this problem, the developer has to replace this direct field access by a call to the appropriate setter method.

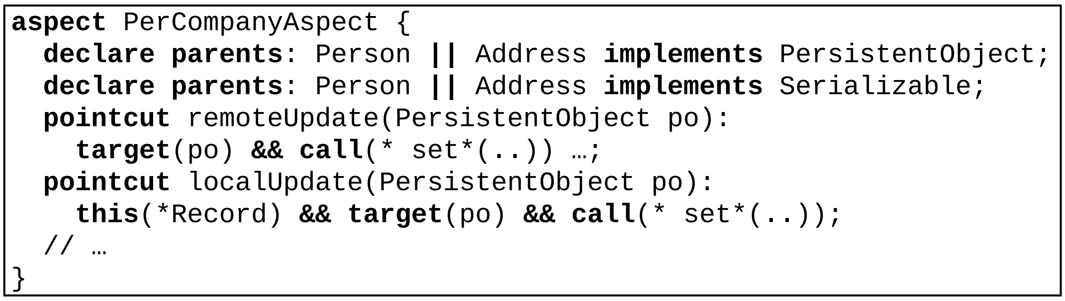


```
// ====== Application-specific aspect ======
public aspect ApplicationDatabaseAccess extends DatabaseAccess {
  declare parents: (Person || Address) extends PersistentRoot;
  pointcut traplnstantiations(): call(PersistentRoot+.new(..));
  pointcut trapupdates(PersistentRoot obj): … &&
   this(obj) && execution(public void PersistentRoot+.set*(..);
  // …
}
// ====== The base code ======
// …
Address a1 = new Address(“Foo Street 45”, “45111”, “Essen”);
Person p1 = new Person(“Mina”);
p1.setAddress(a1);
p1.address = new Address(“Boo Street 10”, “45233”, “Essen”);
```

Figure 3.16: Preparing code for persistence in PAA

3.2.2.6 Code-Level Preparation in DPA

In DPA [Soares et al. (2002)], the situation is exactly the same, because of the use of a similar pointcut specification. As Figure 3.17 shows, the `remoteUpdate` and the `localUpdate` pointcuts select the changes that are made to the persistent object if these changes are made using the corresponding setter methods. Therefore, the application developer must prepare the base code for persistence such as she/he modifies all state change statements to persistent objects so that these changes are made by the appropriate setter methods.

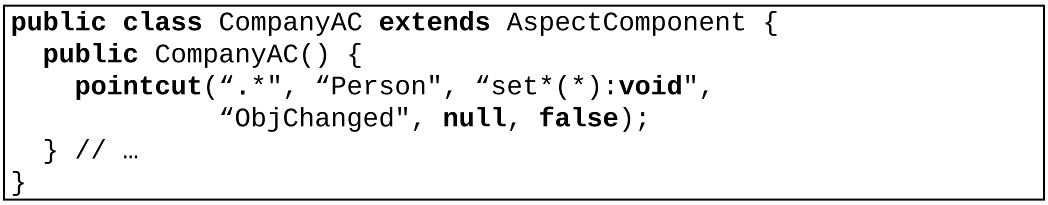


```
aspect PerCompanyAspect {
  declare parents: Person || Address implements PersistentObject;
  declare parents: Person || Address implements Serializable;
  pointcut remoteUpdate(PersistentObject po):
    target(po) && call(* set*(..)) …;
  pointcut localUpdate(PersistentObject po):
    this(*Record) && target(po) && call(* set*(..));
  // …
}
```

Figure 3.17: Selecting persistent field changes based on setter methods requires code preparation in DPA

3.2.2.7 Code-Level Preparation in JAC

The situation with JAC aspect-oriented framework is similar to the above two frameworks. Figure 3.18 shows how the pointcut specification requires that any change to the `Person` objects can be made persistent only if these changes are made by using the setter methods of the corresponding persistence fields (note the `set*` wildcarded method name).



```
public class CompanyAC extends AspectComponent {
  public CompanyAC() {
    pointcut(“.*", “Person", “set*(*):void",
             “ObjChanged", null, false);
  } // …
}
```

Figure 3.18: Selecting persistent field changes based on setter methods requires code preparation in JAC

3.2.2.8 Summary of AO-Systems Code-Level Preparation

The problem with aspect-based persistence frameworks is that they compromise the principle of persistence independence in the same way as in the conventional object-oriented frameworks. Therefore, Figure 3.15 above also depicts this problem.

In this case, the application developers are concerned with ensuring persistence in the base code as well as in the persistence system level. Note that the application developer’s contribution in this situation also includes the preparation of the third-party code classes for persistence (in order to fulfill naming conventions, etc.).[2] This is contradictory to the principle of persistence independence. Hence, the current aspect-oriented persistence solutions fail to fulfill the orthogonal persistence.

[2] This is a result of the so-called *property-based crosscutting* [Gybels and Brichau (2003)].

3.2.3 Initial Remarks

According to the above assessment, current object persistence frameworks fail to comply with the principles of orthogonal persistence with respect to preparing objects for persistence. This assessment covered both conventional object-oriented frameworks as well as aspect-oriented persistence frameworks. It was pointed out that the application base code is not orthogonal to the persistence system. The assessment was divided into two levels as follows:

1. **At type-level.** The developers of the base code still need to prepare the objects at type-level for persistence, which is against the type orthogonality principle because a type can not be persistent unless it is prepared to be so. Moreover, the persistence independence principle is also compromised because the base code types must be prepared to be made persistent, which breaks the base code reusability.

2. **At code-level.** The developers still have to be concerned with the persistence at the code-level, which breaks the persistence independence property. In this case, the code contains specific parts where persistence manipulations should be triggered in both persistence technologies.

It must be mentioned that aspect-oriented persistence frameworks provide better level of orthogonal persistence than the conventional object-oriented solutions. With respect to type orthogonality, the developers in aspect-based systems benefit from the inter-type declaration feature of aspect-orientation to introduce persistence related type enhancements inside the aspect layer, and these modifications are woven into the base code during the compilation process. With respect to persistence independence, aspect-oriented persistence systems do not require the base code to contain persistence utility-based code such as defining persistence managers or sessions as it is the case in conventional approaches (cf. e.g. Section 3.2.2.2). Furthermore, aspect-based systems do not require the base code to contain explicit invocations to persistence operations, e.g. to save or update a persistent object, and transaction demarcations, whereas in conventional systems these invocations are required.

However, despite of the improvement toward a better orthogonal persistence with aspect-oriented approaches, these systems suffer from another serious problem that is called *uniformity* [Kienzle and Guerraoui (2002)]. For example, in AspectJ-based persistence frameworks (DPA and PAA), the selection of the persistent objects is based on their types. This in turn produces general specification of the pointcuts that are responsible for the selection. As a consequence, e.g. in PAA framework, the `trapInstantiations` pointcut selects *all* application objects that are of type `PersistentRoot`, hence, the advice will make them persistent.[3]

A promising solution would be to point the persistence aspects to decide on the selection of the persistent objects based on whether they are reachable from other persistent objects. This solution may not scale well to the `create` persistence operations, however, it would be a way to solve the problem with respect to `retrieve`, `update`, and `delete`. The next section explains this approach in detail and how current aspect-oriented persistence systems can implement it.

3.3 Object Relationships for Persistence in Aspect-Orientation

As mentioned in the last section, to rely on the reachability relationship between objects to determine whether to persist an object of a persistent type is a promising solution for the problem of uniformity in aspect-oriented systems, at least for `update`, `retrieve`, and `delete` persistence operations. In the following subsections, a number of examples are given to illustrate when reachability is required and and they will show how this procedure can be achieved in current aspect-based persistence systems.

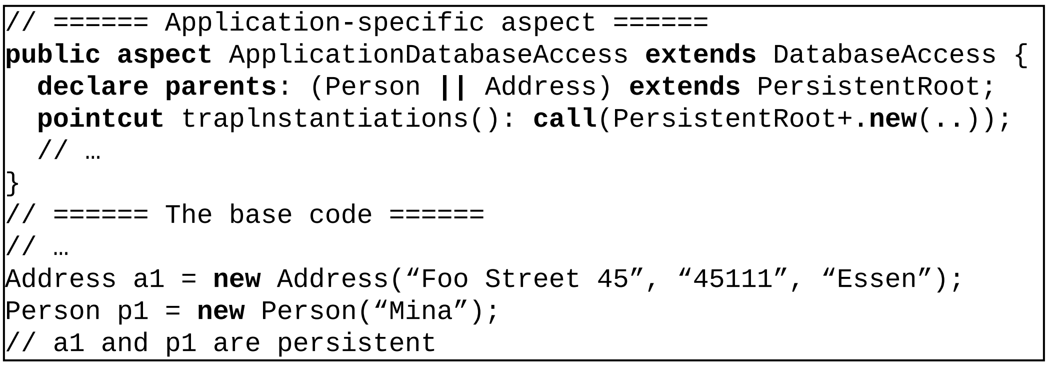
Section 3.3.1 discusses the impact of object relationships in CRUD persistence operations. More complex situations related to object persistence are discussed in Sections 3.3.2 and 3.3.3. Respectively, these situations cover the impact of the

[3] The work in [Kienzle and Guerraoui (2002)] considered the case of selecting the transactional objects, however, the problem of uniformity is applicable to the case of selecting the persistent objects.

non-local object information in aspectizing two different locking-based concurrency control policies, namely, field-based locking and cascading-version locking.

3.3.1 Example 1: Pure Persistence Problem Caused by Uniformity

In Figure 3.19, the concrete pointcut `trapInstantiations` is taken from the aspect `DatabaseAccess` of the previously explained PAA persistence framework. The `Person` and `Address` classes are declared in the `ApplicationDatabaseAccess` aspect to extend the `PersistentRoot` class. Consequently, the `trapInstantiations` pointcut selects all instantiation join points of `Person` and `Address`. The associated advice will persist these objects preventing the developers from using, e.g., `p1` as a transient object unless it's declared `transient`. A developer must thus consider an inverse problem: How to specify individual transient objects? This requires the base code to contain persistence-related statements and declarations, which in turn leads to a situation of preparing the code for persistence (cf. Section 3.2.2).



```
// ====== Application-specific aspect ======
public aspect ApplicationDatabaseAccess extends DatabaseAccess {
  declare parents: (Person || Address) extends PersistentRoot;
  pointcut traplnstantiations(): call(PersistentRoot+.new(..));
  // …
}
// ====== The base code ======
// …
Address a1 = new Address(“Foo Street 45”, “45111”, “Essen”);
Person p1 = new Person(“Mina”);
// a1 and p1 are persistent
```

Figure 3.19: Persisting *all* instantiated objects in AO persistence systems

If the intention of the developer is to use object `p1` as a temporary transient object and she/he did not define this object as `transient`, `p1` will be persisted. If at a later point of time the developer performs a query to know how many `Person` instances the database contains so far, the query will return a wrong number, i.e.

1 instead of 0 in this case. This affects not only the performance by introducing potentially unnecessary persistence operations, but also data consistency.

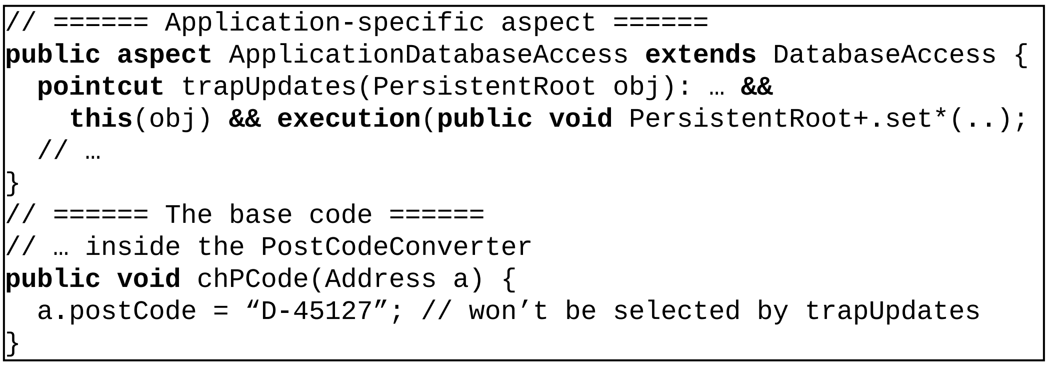
In fact this problem is a consequence of the uniformity characteristic of the pointcut specification in AspectJ as well as in most available aspect-oriented systems. Since `Person` and `Address` types are enumerated in the aspect to extend the `PersistentRoot` class, it is hard to find out when objects of such types are going to be made persistent by examining the code, unless there are other clear signs, e.g. by using the keyword `transient`.

It is clear that selecting the persistent objects at the time of their instantiation is not an easy task, unless the developer is allowed to break the orthogonal persistence by adding explicit proper calls to persist the objects. In general, aspect-oriented solutions cannot help much in the case of making the created objects persistent. However, a possible solution to keep the base code orthogonal to the persistence of object instantiations is to use an interactive tool before the application delivery, which communicates with the developer in an additional test phase to designate the specific situations in the application execution where instantiated objects are to be persistent. Then, corresponding proper pointcuts can be generated or even the required persistence code can be woven directly at those join points. Such solutions are out of the scope of the thesis.

Nevertheless, aspect-oriented programming can act in a better oblivious manner with respect to the persistence operations `update`, `retrieve`, and `delete`. Due to the similarity between these three operations with respect to the designation of the persistent objects, the following discussion covers the case of `update` operation. But first, it is necessary to clarify the problem carefully.

As mentioned earlier, the use of naming conventions determines the selection of object updates join points in current aspect persistence systems. The pointcut specifications in these systems require preparations inside the base code to follow the naming conventions as illustrated above in Section 3.2.2, which is against orthogonal persistence. For example, as Figure 3.20 shows, the `trapUpdates` pointcut in the PAA framework requires that the code is modified to set the persistent objects using setter methods.

As a consequence, direct assignments to persistent objects will not be made persistent. To overcome such objectionable situations, it is necessary to modify



```
// ====== Application-specific aspect ======
public aspect ApplicationDatabaseAccess extends DatabaseAccess {
  pointcut trapUpdates(PersistentRoot obj): … &&
    this(obj) && execution(public void PersistentRoot+.set*(..);
  // …
}
// ====== The base code ======
// … inside the PostCodeConverter
public void chPCode(Address a) {
  a.postCode = “D-45127”; // won’t be selected by trapUpdates
}
```

Figure 3.20: AO persistence systems fail to persist direct updates to persistent objects

the pointcut specification in order to capture all accesses that update a persistent object. Precisely, this can be achieved by using the `set` pointcut designator instead of the `execution` pointcut in the `trapUpdates` pointcut.

Note that using either poincut designators also suffers from the problem of uniformity, unless some other selection criteria are applied such as relying on reachability object relationships. The following is an example of such cases.

Consider the situation where the intention is to persist the root class `Company` of the object model in Figure 3.1.[4] This means that all objects of other classes can be persistent only if they are reachable from a persistent `Company` object.[5] On the other hand, it is important to allow the direct manipulation of such persistent objects from foreign clients. Therefore, these manipulations must become persistent whenever it is required.

In aspect-oriented programming, such a case brings up the question about the available join point properties at object change join point. Some information is required for selecting the join points and respectively, making the corresponding dirty objects persistent. Here, the important object information consists of three parts: The changing object, the changed object, and the fact whether this dirty

[4] This seems to solve the problem of enumerating all persistent classes.

[5] Such a structure is similar to the *second-class objects* in JDO, which have no JDO identity and any change to their state will be conveyed to the owner company object, which will assume the dirty state.

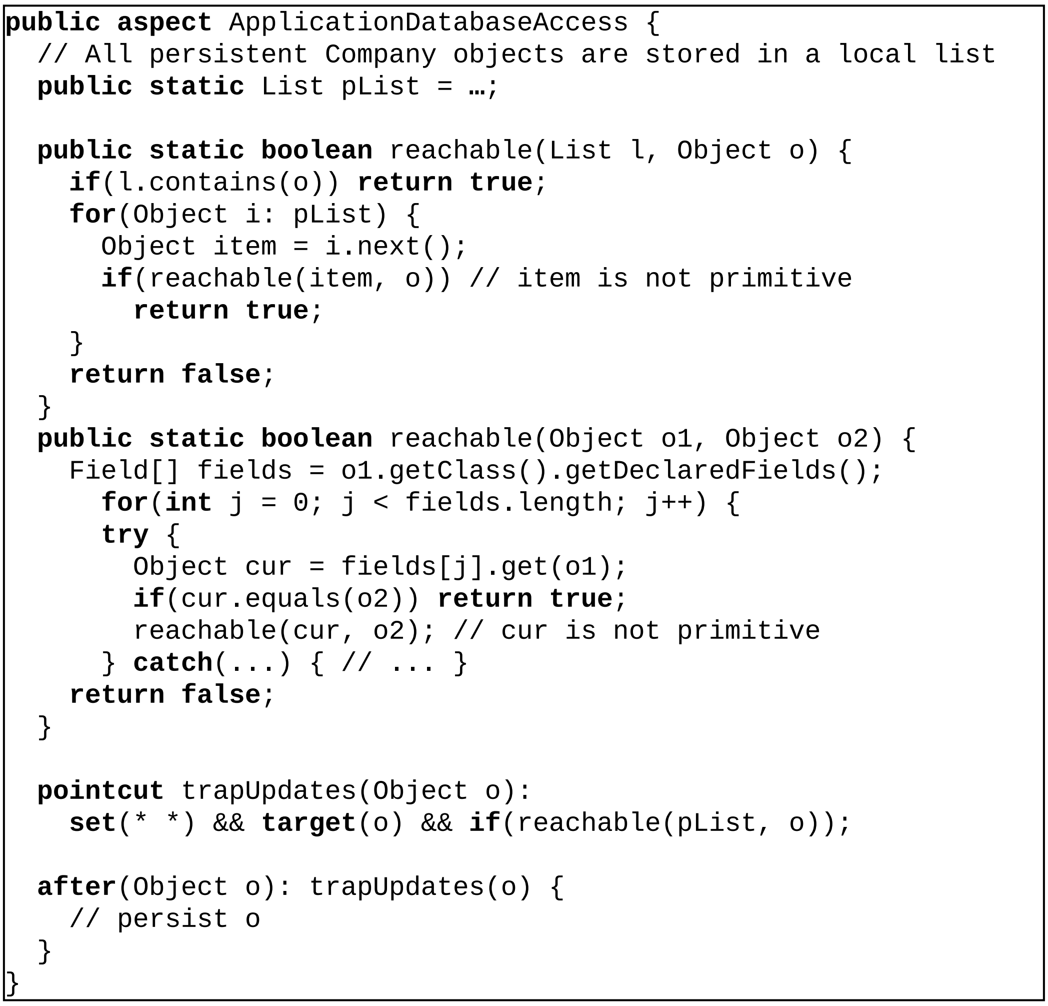
object is reachable from some persistent company object.

Figure 3.20 illustrates an example of such a possible join point. Assume that a `PostCodeConverter` object wants to change the `postCode` field of the `Address` object called `a`. The method `changePostCode` gets `a` as a parameter and directly assigns a new post code to it. Accordingly, the object `a` must be changed if it is part of a persistent `Company` object. The new `trapUpdates` pointcut specification in the figure selects all such join points, where the `target` pointcut provides access to the changed object `a`. The missing part is to decide whether this address object is reachable from some persistent company object, which in turn requires access to all persistent company objects in the heap. Unfortunately, these parts of the heap are not available at the local context of such join points. Moreover, current aspect-oriented languages do not provide pointcut constructs that could be used to get access to this non-local object relationships. Hence, to solve this problem, one is obliged to code by hand workarounds that bring the access to the required non-local object information.

In Figure 3.21, assume that the list `pList` maintains all persistent company objects. The method `reachable` is used to decide whether the second parameter `o` is reachable from `pList`, i.e. it returns `true` if object `o` is part of the object closure of a given persistent `Company` object, otherwise it returns false. The method works in a simple way to traverse a subgraph of the object graph. The root of the target subgraph is the `pList` object.

It is clear that this implementation of the algorithm, when applied to an object graph with cycles, will not terminate. Therefore, in order to simplify the example, it is assumed that the `pList`-rooted subgraph represents a tree, i.e. contains no cycles. The method runs through all non-primitive fields of the current object `cur`, extracts the object represented by the current field and then returns `true` if this object equals `o`. Otherwise, it calls recursively itself with that object. The `trapUpdates` is modified to select all set operation join points whose target is `o` and if `o` is reachable from the list `pList`. The associated `after` advice simply persists the target object of the selected set join point.

From this example, it is obvious that the developer is obliged to code complex workarounds to get access to the non-local part of the join point context that is



```
public aspect ApplicationDatabaseAccess {
  // All persistent Company objects are stored in a local list
  public static List pList = …;

  public static boolean reachable(List l, Object o) {
    if(l.contains(o)) return true;
    for(Object i: pList) {
      Object item = i.next();
      if(reachable(item, o)) // item is not primitive
        return true;
    }
    return false;
  }
  public static boolean reachable(Object o1, Object o2) {
    Field[] fields = o1.getClass().getDeclaredFields();
      for(int j = 0; j < fields.length; j++) {
      try {
        Object cur = fields[j].get(o1);
        if(cur.equals(o2)) return true;
        reachable(cur, o2); // cur is not primitive
      } catch(...) { // ... }
    return false;
  }

  pointcut trapUpdates(Object o):
    set(* *) && target(o) && if(reachable(pList, o));

  after(Object o): trapUpdates(o) {
    // persist o
  }
}
```

Figure 3.21: Modified pointcut that supports objects reachability

based on object relationships, here, the reachability relationship. Such complex workarounds are undesirable solutions for several reasons:

- First, the pointcut specification used in this example does not reflect the semantics of join point selection. That is clear since the selection depends on hand-coded routines instead of using pointcut constructs.
- Second, these solutions suffer from performance problems due to the use of reflective facilities provided by the programming language.
- Third, these solutions are not easy to comprehend and maintain. However, one can provide a sort of generic API that can be used directly by the application developer.

3.3.2 Example 2: Field-Based Locking Mechanism

In locking-based concurrency control literature, a large number of researchers discuss *locking granularities* [Gray et al. (1975)], propose techniques for *fine-granularity locking* [Mohan and Haderle (1994); Panagos et al. (1996)], and discuss the benefits and the effects of *multiple locking granularities* [Ries and Stonebraker (1977)]. The granule of the data that can be locked is the whole database, a set of objects, an object, or a field of an object [Kemper and Moerkotte (1994)]. This section focuses on locking the fields of the object that are being changed, so that multiple transactions can work on this object concurrently.

In Figure 3.22, the `Company` instance `c`, which references the object `pres`, is added to the persistent list `p`. Hence, this company object and all objects in its closure are to be made persistent. Assume that the developer is interested in applying locking on the fields of the `Employee` object. From the figure, there are two separate concurrent threads that attempt to update the state of the employee. The first thread (named *Thread1*) in `pcc`, wants to change the `postcode` value of object `addr` that is part of object `pres`. The second thread (*Thread2*) in `pm`, attempts to change the `phone` field of the employee. In order to allow both threads to modify the employee object simultaneously, one should acquire separate write locks for the fields `residence` and `phone` rather than for the object `pres`.

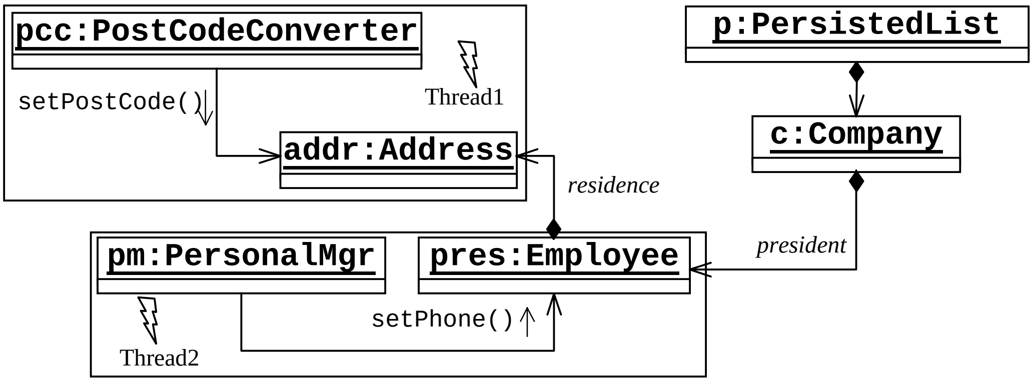


Figure 3.22: Two separate concurrent transactions attempt to change an `Employee` object

In aspect-oriented terms, each update is a join point that should be selected since the employee object belongs to the persistent list `p`. The needed information is: the objects `p` and `pres`. This information is necessary to determine the reachability between both objects. The aspect then should acquire a write-lock for each field that is to be changed. Here, the needed information is the fields `residence` and `phone`.

The local relevant context at the join point in *Thread1* is object `addr`, and the local relevant context for *Thread2* is object `pres`. In *Thread1*, objects `p` and `pres`, in addition to the field name `residence`, are considered to be non-local, while in *Thread2* the non-local relevant information is object `p`.

Similar to the case of persistence operations in the last subsection, in order to get access to this non-local information, the developers are required to hand-code some workarounds that participate in selecting the join points.

Figure 3.23 shows the pointcut definitions in a possible implementation of the required field-based locking concurrency control mechanism using AspectJ. The aspect `FLConcurrencyControlAspect` maintains the persistent lists in a `List` object called `allPLists`. The pointcut `persistentLists` selects the instantiation join points of `PersistentList` objects. The pointcut `objChg` selects all set join points where the target object is `o` of the general supertype `Object`. The methods `reachable` are taken from Figure 3.21.

```
public aspect FLConcurrencyControlAspect {
  // all persistentList objects are stored locally in allPLists
  List allPLists = new LinkedList();

  pointcut persistentLists(PersistentList list):
    execution(new(..)) && target(list);

  pointcut objChg(Object o): set(* *) && target(o);
```

Figure 3.23: FLConcurrencyControlAspect of field-locking: Pointcut specifications

Figure 3.24, on the other hand, shows the advice code of the concurrency control aspect. The `after` advice adds the persistent lists to the aspect local variable `allPLists` at the time of their instantiation. The `before` advice first iterates throughout the list of the persistent lists in the `allPLists` and checks whether the target object `o` is reachable from any persistent list (may be from more than one list).

In fact, the determination of the reachability relationship depends on non-local object information to the set join point. If `o` is reachable from a given persistent list `plist`, which means that `o` is a persistent object, the advice checks if `o` is of type `Employee`. If that is the case, the changed field is determined easily using the introspective facility of AspectJ on the reference variable `thisJoinPoint`. Finally, the advice acquires a lock on this field.

On the other hand, if the object `o` is not of type `Employee`, the advice iterates throughout all elements of the `plist`. If `o` is reachable from a given item in `plist` then the advice invokes the method `getFirstEmployee` that returns the first `Employee` object `e` that references `o`. If `e` is not `null` then the advice locks each field of `e` that references `o`. Note that the choice of getting the first owner `Employee` object in this solution is taken for simplicity reasons, since there might be more complex situations where a given object is referenced from more than one `Employee` objects that are part of the same reference path in the object graph.

The pointcut `objChg` of Figure 3.23 selects all set join points whose target is `o` of type `Object`. According to the object graph in Figure 3.22, the pointcut selects both set join points in a sequence depending on the order in which both

```
// to store all persistentList objects in allPLists
after returning(PersistentList list): persistentLists (list)
{ allPLists.add(list); }

// to acquire locks for dirty fields in the Employee objects
before(Object o): objChg(o) {
  // check for reachability from a persistent list
  for(List plist : allPLists)
    if(reachable(plist, o)) {
      if(o instanceof Employee) {
        String fname = thisJoinPoint.getSignature().getName();
        // get the field by using its name and lock it
          Field field = …
      } else {
        // get the Employee object e that o is reachable from
        for(Object item : plist)
          if(reachable(item, o)) {
            Employee e = getFirstEmployee(item, o);
            if(e != null) {
              Field[] flds = e.getClass().getDeclaredFields();
              for(int j = 0; j < flds.length; j++) {
                try {
                  Object cur = flds[j].get(e);
                  if(reachable(cur, o))
                    // lock field flds[j]
                } catch(...) { // ... }
}   } }   } } }

public Employee getFirstEmployee(Object src, Object des) {
  if((src instanceof Employee) && reachable(src, des))
    return src;
  // for each field of src do
  getFirstEmployee(field.get(src), des);
}
```

Figure 3.24: FLConcurrencyControlAspect: Advice code

threads occur. However, this order is not relevant in this case because the main task is to lock the fields that are going to be changed, if they are not locked already.

For the change operation in the `phone` field of the `pres` object (*Thread2*), the pointcut `objChg` binds the `pres` object to the variable `o`, which is exposed to the `before` advice in Figure 3.24. Since `pres` is reachable from the persistent list `p`, and it is of type `Employee`, the advice will lock the `phone` field.

In *Thread1*, the set operation on the `Address` object `addr` is selected also by the pointcut `objChg`. Then, the advice must again decide whether to adapt this join point or not. First, the advice will figure that the target object `addr` is reachable from the list `p`. Then, since `addr` is not an instance of `Employee`, the advice will search for the first `Employee` object that owns the `addr` object, if any. Finally, the advice will find that `addr` is reachable from the employee object `pres` through the field `residence`, which will be locked by the advice.

From this example, the target object of the set operation could be of any type. The reason for such general pointcut specification is that it is required to select every change to the state of an `Employee` object or any object in its reference closure. Accordingly, the pointcut `objChg` selects *all* state changes in any object. Then, it is the task of the advice to determine whether to continue selecting the given field set join point in order to adapt it or not. This means that the advice, which is supposed to be a construct that provides join point adaptation, is also responsible for the join point selection. This is considered to be a *mixture* between the semantics of join point selection and join point adaptation mechanisms.

The rationale behind this problem is that AspectJ as well as current aspect-oriented systems do not provide pointcut constructs that can abstract over the non-local join point properties that are based on object relationships. In order to solve this problem, developers provide workarounds to get access to the non-local object relationships. As shown in this simplified example, such solutions suffer from many problems: the solutions are problem-specific, complex, difficult to maintain, and error-prone. Moreover, their code does not reflect the semantics of join point selection and adaptation.

3.3.3 Example 3: Cascading Version Locking Mechanism

In order to solve concurrency control problem, researchers also proposed a number of version-based locking policies [Kim and Park (1998); Lin and Nolte (1983); Mohan et al. (1992)]. In these policies, all transactions can grant shared read access to the object, and whenever a transaction attempts to update the state of the shared object, the application should check whether this update is performed on the right version of the object.

Version locking mechanisms use a so-called version (or write-lock) field that is added to every object and compare this field every time an update operation on the object is committed with the current value in the datastore. If they are equal, the change is committed to the datastore, otherwise, the change is disallowed. This indicates that the object must have been updated by another transaction. In the cascading version locking, the version field of all objects that reference the dirty object must be updated also.

As an example, consider the object graph in Figure 3.25. Two `Company` instances, `com1` and `com2` are referencing the same object `addr` that is being updated by the `PostCodeConverter` object `pcc`. The `Company` instances are stored in the persistent list `p`. According to the version locking policy, any change to the `addr` will update the version field of `addr` as well as the version fields of its owner objects, i.e., `com1` and `com2`.

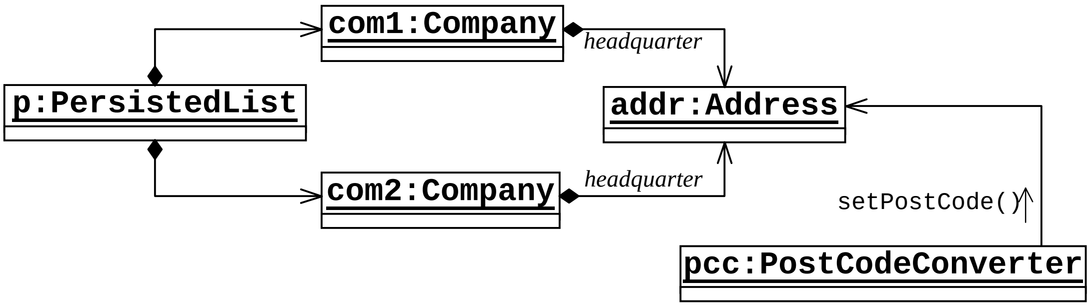


Figure 3.25: A shared Address instance between two Company instances being changed

The aspect's task would be to select the join point where the object `addr` is being changed and to check whether this object is reachable from any persistent list. If so, the aspect should perform the dedicated version locking policy by checking and accordingly updating the version fields of the updated object as well as of its owners.

```
public aspect VLConcurrencyControlAspect {
  List allPLists = new LinkedList();
  pointcut persistentLists(PersistentList list):
    execution(new(..)) && target(list);

  pointcut objChg(Object o): set(* *) && target(o);

  public boolean reachable(List l, Object o) { ... }
  public boolean reachable(Object o1, Object o2) { ... }

  // acquire locks for dirty fields of the Employee object
  after(Object o): objChg(o) {
    // check for reachability from a persistent list
    for(List plist : allPLists)
      if(reachable(plst, o)) {
        // get the Employee object e that o is reachable from
        for(Object item : plist)
          if(reachable(item, o)) {
            checkVersion(item, o);
            // check and update the version field of object o
            // and commit the transaction
          }
      }
  }

  public void checkVersion(Object src, Object des) {
    // check and update the version field of the object src
    Field[] fields = src.getClass().getDeclaredFields();
    for(int j = 0; j < fields.length; j++) {
      try {
        Object cur = fields[j].get(src);
        if(reachable(cur, o)) // cur must not be primitive
          checkVersion(item, o);
      } catch(...) { // ... }
    }
  }
}
```

Figure 3.26: VLConcurrencyControlAspect

Figure 3.26 shows a possible implementation of the above-described locking policy. In this case, the aspect `VLConcurrencyControlAspect` is also making use of the aforementioned object `allPLists` and method `reachable` in order to get access to the non-local object information at the set join points. The pointcut `objChg` has the same semantics as the corresponding one in Figure 3.23, hence, it selects all set join points where the target is an object `o` of type `Object`. The associated `after` advice, then, is responsible for checking the reachability between the persistent lists and the updated object. It will traverse the object graph, with the help of the recursive method `checkVersion`, starting from the persistent lists in order to find the owner objects of the updated one. Then it will check and update their version fields and determine whether to commit or to reject this update.

According to the collaboration diagram in Figure 3.25, the set operation of the object `addr` is selected by the pointcut `objChg`. The later binds `addr` to the variable `o` and exposes this context to the `after` advice. Then this advice will find that `o` is reachable from the list `p`. Hence, it will invoke the method `checkVersion`. This method traverses the whole part of the object graph that starts from `p`, however, it avoids unnecessary paths by examining the reachability at each node in the object graph. Whenever an owner object of the object `o` is found, the method decides whether to update the version field or not.

From this example, it is obvious that one cannot avoid a complex implementation of the cascading version-based locking policy. As mentioned in the previous example, the only way currently available for the developer in current aspect-oriented systems is to apply introspective facilities of the language to traverse the entire reference path to get the required accesses.

These kind of solutions are not trivial, error-prone and mostly not reusable. In addition to that, these solutions are mixing the semantics of the join point adaptation and the join point selection mechanisms, since the advice participates in the selection of the join point by examining the non-local reachability object information as well as providing access to all relevant non-local objects. Both of these functionalities are supposed to be part of the pointcut language.

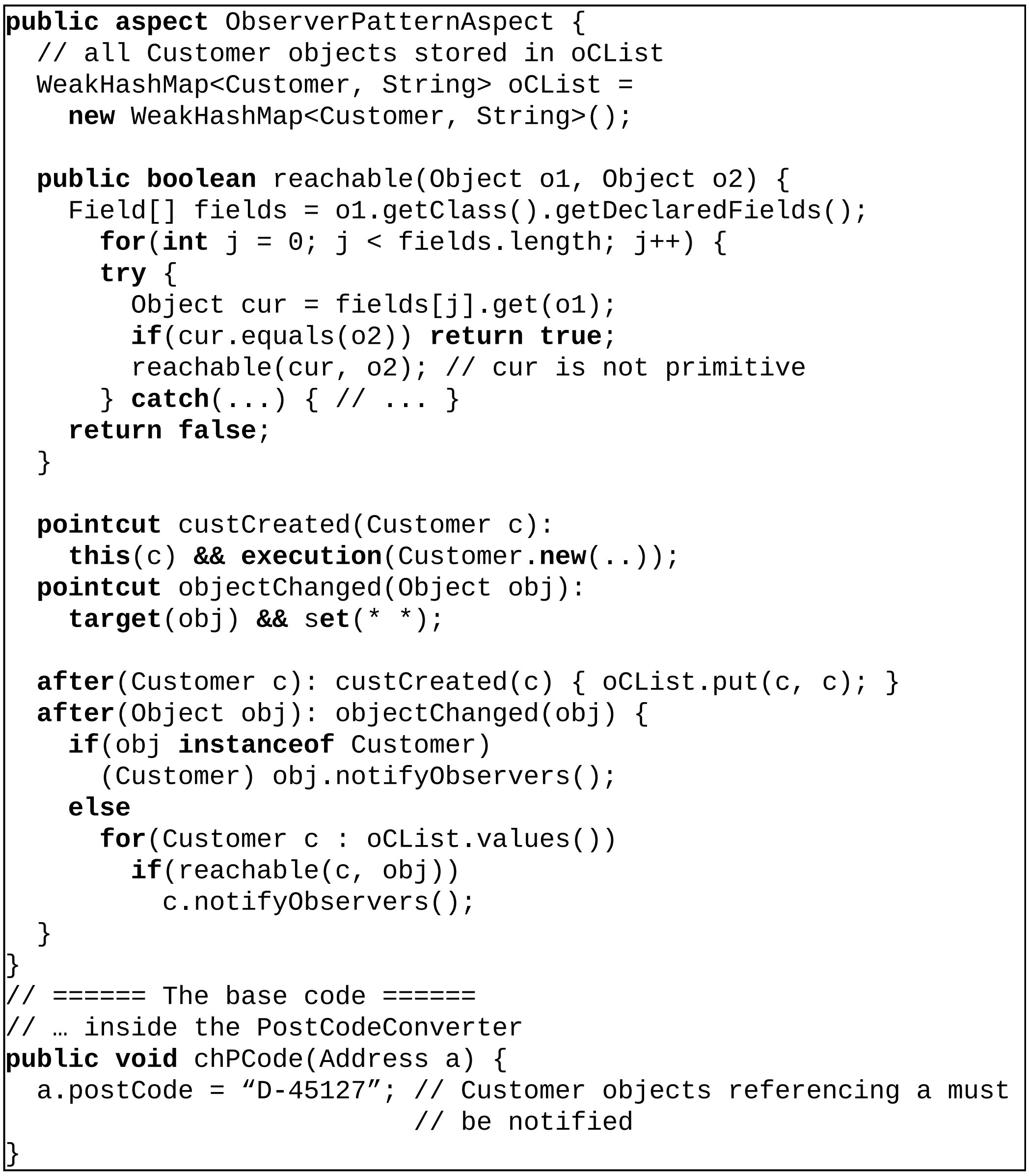
3.4 Object Relationships for other Concerns in Aspect-Orientation

This section continues the argumentation presented in the previous section about the importance of the object information for aspect-oriented programming. The following discussion is related to aspectize a different crosscutting concern, namely the *observer design pattern* [Gamma et al. (1994)]. The implementation of the observer design pattern is considered to be a typical case of crosscutting concerns in the aspect-oriented literature [Gybels and Brichau (2003); Hanenberg (2005); Ostermann et al. (2005); Stein et al. (2002); Veit and Herrmann (2003)].

The example presented here is based on the object model in Figure 3.1 above. These associations between the classes represent ownership relationships, which are unidirectional relationships. Therefore, the information about such relationships is available at the owner side, e.g. `Customer` objects are aware of the relationship to their associated `residence` addresses, but the inverse is not true.

The observer design pattern in this example is assumed to be applied to `Customer` objects. The subjects are the `Customer` objects, while they are observed by a given set of observers representing the user interface objects. That means, any state change to a `Customer` object should send a message `notifyObservers` to this changed `Customer` object, hence, notifying the observers that in turn reflect the changes to the user interfaces. The state of the `Customer` object is its whole object closure. That means, even if the change is happening to an object that resides in the object closure of a given `Customer` object, the later must be notified.

In terms of aspects, capturing the changes on any object can be achieved by using the `set` pointcut designator. Then, it is necessary to check if this change is performed directly on a `Customer` object, or on any of its referenced objects, in order to notify the corresponding observers about this change. If the change was applied to a customer object directly, the advice responsible for performing the notification can easily access this customer object by means of the context exposure pointcut `this`. However, if the change happens to some other object that is part of the object closure of a given `Customer` object, it would be difficult for the advice to access the owner customer to notify its observers. Again, the reason is that current pointcut languages do not provide a construct that can expose this non-local object information. Similar to the above situations, the developers are forced to write the required code by hand.



```
public aspect ObserverPatternAspect {
  // all Customer objects stored in oCList
  WeakHashMap<Customer, String> oCList =
    new WeakHashMap<Customer, String>();

  public boolean reachable(Object o1, Object o2) {
    Field[] fields = o1.getClass().getDeclaredFields();
      for(int j = 0; j < fields.length; j++) {
      try {
        Object cur = fields[j].get(o1);
        if(cur.equals(o2)) return true;
        reachable(cur, o2); // cur is not primitive
      } catch(...) { // ... }
    return false;
  }

  pointcut custCreated(Customer c):
    this(c) && execution(Customer.new(..));
  pointcut objectChanged(Object obj):
    target(obj) && set(* *);

  after(Customer c): custCreated(c) { oCList.put(c, c); }
  after(Object obj): objectChanged(obj) {
    if(obj instanceof Customer)
      (Customer) obj.notifyObservers();
    else
      for(Customer c : oCList.values())
        if(reachable(c, obj))
          c.notifyObservers();
  }
}
// ====== The base code ======
// … inside the PostCodeConverter
public void chPCode(Address a) {
  a.postCode = “D-45127”; // Customer objects referencing a must
                          // be notified
}
```

Figure 3.27: Implementing observer pattern in AOP for customer objects

A possible implementation for such an observer pattern in terms of current aspect-oriented systems is depicted in Figure 3.27. The WeakHashMap is used to maintain all instantiated Customer objects, which are added by the pointcut custCreated and the associated after advice.[6] The pointcut objectChanged selects all change join points to any object. It passes the changed object obj to the corresponding after advice by means of this pointcut. The advice checks if obj is of type Customer. If so, the advice notifies the object observers, otherwise, the advice iterates through the list of available customer objects and checks if this modified object is reachable from any customer object. The advice notifies all customers of the change of their owned object.

As stated in the previous sections, the use of complex workarounds here results in a number of problems. The most difficult one is that such solutions suffer from a conceptual problem since they mix the semantics of advice and pointcuts. Here, the advice checks whether the modified object is reachable from a Customer object, if so, the change join point is selected. This kind of mixing produces pointcuts that are not expressive and do not reflect the semantics of join point selection.

3.5 Problem Statement

In order to provide orthogonal object persistence, any persistence solution must fulfill the three principles of orthogonal persistence: type orthogonality, persistence independence, and transitivity. Orthogonality between the application base code and its persistence requires that the code does not contain signs at which persistence manipulation should take place. This exactly meets the obliviousness property of aspect-oriented programming: The code must not be prepared for persistence. The thesis distinguishes between two kinds of base code preparation: The preparation at the type-level and the preparation at the code-level.

The assessment presented in this section considered current solutions for object persistence from object-oriented and aspect-oriented domains in order to show the extent to which these solutions fulfill orthogonal persistence principles. Object-oriented systems are included here for reasons of comprehension due to

[6] The use of the WeakHashMap allows the customer objects to be garbage-collected.

their widespread usage in the world applications. The object-oriented systems considered are EJB [SUN (2006a)], JDO [Roos (2002)], and Hibernate [Bauer and King (2005)]. The aspect-oriented systems are the distribution and persistence (DPA) framework by Soares et al. (2002), the persistence framework (PAA) of Rashid and Chitchyan (2003) and the JAC framework [Pawlak et al. (2004)].

It has been shown that current conventional object-oriented systems as well as aspect-oriented frameworks for persistence do not comply with the orthogonal persistence principles as follows:

1. **At type-level:** these systems defeat the type orthogonality and persistence independence principles.

2. **At code-level:** these systems defeat the persistence independence principles.

However, it must be noted that current aspect-oriented persistence systems provide better level of orthogonality since aspects will contain some persistence-related code instead of tangling this code inside different modules in conventional solutions. For example, there is no need to change the type hierarchy of the base code since this is shifted to the aspect code layer. Nevertheless, this solves a part of the problem since aspect-based frameworks require persistent types to be modified to fulfill certain naming conventions.

Likewise, at code-level aspect-based frameworks provide better transparency, since the base code does not contain explicit invocations of the persistence operations and these places will be selected by means of the join point selection mechanism.

However, the selection criteria in the pointcuts provided by current aspect-oriented solutions depends on certain naming conventions such that the application developer must be sure that any persistent object is accessed by using the setter and getter methods. This means that direct accesses to a potential persistent object will not be selected and the persistence manipulations will not take place. Moreover, all accesses to any object of a persistent type by means of setter and getter methods will be made persistent, which may lead to an inconsistent database. This is in fact a direct consequence of the uniformity problem that current aspect-oriented programming systems fail to solve.

Then, by means of illustrative examples, this section showed how object relationships are suitable for solving this shortcoming. With respect to preliminary persistence operations, i.e. create, retrieve, update, and delete (CRUD), object relationships can be used to overcome the uniformity problem in persisting update, retrieve, and delete operations. The solution is to consider the selection of persistent objects based on reachability from another persistent object. Henceforth, a possible solution was provided, where the pointcuts designate persistent objects with the help of workaround code that determined the reachability. Such solutions suffer from several problems:

- They do not reflect the semantics of join point selection,
- they introduce performance problems, and
- they are difficult to comprehend and maintain.

In order to solve such problems, it would be better, if this functionality is integrated as a part of the underlying aspect-oriented programming language. Unfortunately, current pointcut languages do not provide suitable constructs that give access to the non-local reachability relationship.

Moreover, two more examples of concurrency control mechanisms, which is an issue of object persistence, were given to prove the importance of object information for persistence in aspect-orientation. In addition to that, another example about another crosscutting concern, namely the observer design pattern, was presented to strengthen the argument. Those examples show that current aspect-oriented systems must not only have means to address non-local reachability at the level of the programming language. However, they also must have means to provide access to other kinds of non-local object information such as field information or all objects that occur in specific reference paths in the object graph.

3.6 Chapter Summary

This chapter provided a detailed discussion about how current aspect-oriented systems suffer from the lack of supporting non-local join point properties that are based on object information. All the motivating examples that were presented in this chapter are taken from the domain of object persistence.

The assessment discussion illustrated how the available aspect-based solution for object persistence fail to comply with the principle of orthogonal persistence. This assessment was divided into type level and code level.

This assessment showed also some complex situations where the available pointcut language cannot provide elegant and easy solutions. This is because accessing the non-local information is not a trivial task. In order to solve this problem, the developers have to write very complex routines and workarounds that indeed mix the semantics of the join point selection and the join point adaptation mechanisms. Hence, there is a need for abstractions in aspect-oriented programming that provide the required access to the non-local object information.

Chapter 4

Path Expression Pointcuts

As demonstrated in the last chapter, current aspect-oriented persistence frameworks fail to comply with the principles of orthogonal persistence. The rationale behind this fact is that these frameworks break the obliviousness property of aspect-oriented programming since the base code must be prepared for persistence at the type level as well as the code level. Moreover, these systems suffer from the problem of uniformity, whereby it is difficult to designate the persistent objects and consequently affect the data consistency.

In order to solve the second problem, object reachability relationship is shown to be a good candidate to be used to designate persistent objects with respect to retrieve, update, and delete persistence operations. Instead of relying on certain naming convention by restricting accesses to objects with the help of setter and getter methods, persistent objects are selected based on the fact whether they are referenced from other persistent objects. Unfortunately, reachability is, in most pointcut languages, a non-local join point property that can not be addressed. Moreover, it has been illustrated in the last chapter that other persistence related issues as well as other crosscutting concerns require not only the access to non-local object reachability information, rather, there is a need to provide access to other kinds of non-local object information such as field information or even certain parts of the object graph.

Henceforth, developers are obliged to write their own code that provides access to this non-local property in order to decide on the selection of the join points where persistence manipulations must take place. This in turn produces pointcut specifications that are not expressive and do not reflect the semantics of the join point selection mechanism. In order to overcome this undesirable conceptual as well as complexity problem, there is a need for a good abstraction that is expressive, easy to use, and provides the required access to the non-local parts of the object graph.

This chapter is about proposing such a pointcut construct that solves this problem by means of applying the well known *path expressions* [Campbell and Habermann (1974)] in aspect-oriented programming. The new pointcut is called *path expression pointcut* (PEP). Throughout the remaining text, the thesis refers to path expression pointcut either by its full name, by its abbreviation (PEP) or shortly by path pointcut.

The structure of this chapter is organized as follows. In Section 4.1, an introduction to path expressions is presented. Section 4.2 introduces informally the different concepts of PEP: The concrete syntax and semantics using illustrative examples in order to show all facets of the concept. This will include the PEP's support of pattern matching, its mechanisms of parameter bindings and context exposure, how it modifies the mechanism of advice execution, and how it makes the non-local relevant information to the join point available for the aspect.

Section 4.3 discusses some issues regarding the informal description of PEP. Moreover, it discusses some typing issues of PEP.

Finally, a short summary for the chapter is given in Section 4.4.

4.1 Path Expressions

Path expressions were first introduced by Campbell and Habermann (1974) in order to synchronize the operations on data objects by defining the sequence in which these operations must be executed. Then, this technique became accepted as a concise syntactical means to reference objects. This is why it became a central ingredient of object-oriented query languages such as the EJB query language (EJB-QL) [Sarang et al. (2001); SUN (2006a)] and the JDO query language [Roos (2002)]. The technique was applied by Frohn et al. (1994); Kifer et al. (1992) in querying the objects in object-oriented databases. Henrich and Robbert (2001) used path expression in a query language for structured multimedia databases.

The idea behind using path expressions in querying object-oriented databases is that the object relationships can be considered as sequences of objects $o_1, o_2, \ldots, o_n$, where $n \geq 0$. This sequence is called a database path [Kifer et al. (1992)]. To express such sequences, a path expression of the form $sel_0.attr_0\{[sel_1]\}, \ldots, attr_m\{[sel_m]\}$ was used, where sel_i is a variable that ranges over objects of a given type or a direct object identifier, and $attr_i$ is either an attribute name or attribute variable that ranges over a set of attribute that belong to an object. More general notations and wildcards could be used to express more efficient queries. The usage of path expressions simplifies object queries. For example, consider the following query:

```
SELECT y FROM Person x WEERE x.residence[y].city['Essen']
```

According to the object model given in Figure 3.1, the variable `x` ranges over objects of type `Person`, each has a `residence` field of type `Address`, so, `y` ranges over objects of type `Address`, and the selected addresses should be in the Essen city.

Then, the W3 Consortium used path expressions in querying the XML databases by adding them to the XPath [Clark and Derose (1999)] and [Boag et al. (2007)] XQuery languages. The idea was the same as the one in object-oriented databases, i.e., a path represents a sequence of the XML elements and their attribute. Here, the objects are nodes in the XML document tree and special notations are used to determine these nodes, e.g., in XQuery, '`/`' denotes the root node from where the search should start or a separator between steps in the path (parent-child relations), where '`//`' denotes the descendants of the current node. For example, the following is a query that selects the addresses in Essen city, which belong to a person:

```
document("company.xml")/person//address[city = "Essen"]
```

This flexibility and expressiveness make path expressions a stronger candidate to be used in addressing object relationships in programming languages also. For example, adaptive programming [Orleans and Lieberherr (2001)] and strategic programming [Lämmel et al. (2003)] use path expressions to define the traversal strategies that are used to traverse object graphs at run-time. Similar notations to the above ones are used here also such as the "`*`" wildcard that means any object of any type. For example, the following strategy determines that the traversal algorithm should start the traverse from the objects of type `Company` until it reaches an object of type `Address` through `Person` objects:

```
from Company via Person to Address
```

Consequently, addressing object relationships in aspect-oriented language can be achieved by using path expressions. This means that non-local object information from the object graph at a given join point can be expressed by means of path expressions. This is what the rest of this chapter will explain.

4.2 Introduction to Path Expression Pointcuts

This section gives an informal description of the path expression pointcuts using some illustrative examples that are based on the object model of Figure 3.1. This introduction includes the syntax of the path pointcut in Section 4.2.1 and its semantics in Section 4.2.2.

A number of issues and concepts of aspect-oriented programming are affected due to the introduction of path expression into pointcut languages. These issues cover the extensions to the parameter bindings, context exposure, and advice execution mechanisms of aspect-oriented programming. Moreover, the new construct also supports the mechanism of pattern matching, which is used in most aspect-oriented languages. All these issues are going to be discussed in Section 4.2.2.

4.2.1 Syntax

Path expression pointcut (PEP) is an explicit extension to the pointcut language. That means, it is a pointcut construct that can be used to define pointcuts in aspects as it is the case with other pointcut designators. PEP takes two parameters; the first parameter is an object of the so-called *path expression graph* (`PEGraph`) type, which represents the resulting graph from the matching process.

The second parameter is the *path expression pattern* that specifies the relationships between the given objects and is used as the matching criteria. The general syntax of PEP is:

```
path(PEGraph pg, PathEpressionPattern);
```

The concrete syntax of PEP is given as follows:

PathExpressionPointcut Syntax

```
PEP           ::=  “path”“(” PGT id “,” PathPattern “)”
PathPattern   ::=  (ObjPattern “ – ”FieldPattern“ → ”)+ ObjPattern
ObjPattern    ::=  TypePattern id | id
FieldPattern  ::=  IdPattern | “/” | “ * ”
IdPattern     ::=  “ * ” [IdPattern] | id “ * ” | id
TypePattern   ::=  Defined according to the AspectJ syntax
Id            ::=  Defined according to the Java syntax
PGT           ::=  PEGraph | PEGraph<Id, PGT>
```

The `PEGraph` object should be included in the pointcut header. This parameter announcement can be overridden by replacing the `PEGraph` stated in the pointcut header with a subtype of it in the PEP.

Similarly, the types that occur in the path pattern may override corresponding types mentioned in the pointcut header such that the former types are subtypes of the later ones. As it will be explained later, PEP supports resolving the actual types. In case the types are not included in the path pointcuts for the object variables, the path pointcut will consider their corresponding types that appear in the pointcut header.

The path patterns consist of two parts. The first is at least one object pattern followed by a relation. The second part is an object pattern. The object pattern can be an identifier that represents an object variable or a type pattern as defined in AspectJ followed by the object variable. The relationship is surrounded by the tokens ”`-`” and ”`->`” with a field pattern in between. A field pattern is either an identifier pattern, a ”`/`” or a ”`*`”. The identifier pattern can be a simple Java-like identifier or a sequence of valid identifier characters that contains ”`*`”. Finally, the path expression graph ”`PEGraph`” is either parameterized or a type-erased "PEGraph". For the purpose of simplicity, from this point on, all presented examples are making use of the unparameterized "PEGraph" type until the interface of this type is discussed later on.

For example, the following is a valid complete path pointcut specification, where Company, Person+ and *Address are type patterns, c, p, a and s are object identifiers, and *, / and str* are relations.

```
path(PEGraph g, Company c -*-> Person+ p -/-> *Address a –str*-> s)
```

The corresponding meaning and semantics of the PEP syntax are explained in the following.

4.2.2 Semantics

The path pointcut traverses the current object graph in order to find paths that match a given path expression. A pointcut that uses a path pointcut picks out the join points where there exists at least one matching path. The path pointcut calculates the path expression pattern against the current heap and adds all matching paths to a generated PEGraph object. When the evaluation process ends, the resulting PEGraph object is bound to the variable name specified at the first parameter of the path pointcut. This parameter will be added to the resulting parameter bindings being exposed to the pointcut and the associated advice.

The path expression pointcut can be directly added to the syntax of the current AOP languages, e.g. AspectJ syntax, by declaring path expressions as pointcuts. Consequently, this permits to compose path pointcuts using the operators "&&", "||", and "!". For instance, the pointcut in line 1 of Figure 4.1 selects the method execution join points whose method names start with the prefix set and where there is at least one matching path between a Company object c and an Address object a.

While the pointcut in line 2 of Figure 4.1 selects the field set join points where there is no matching paths in the object graphs from c to a.

In order to reduce the number of matching paths, one can compose more than one path pointcuts using these operators, e.g. the pointcut in line 3 of Figure 4.1

```
1. execution(* *.set*(..)) && path(p, Company c -/-> Address a)

2. set(* *. *) && ! path(p, Company c -/-> Address a)

3. get(* *.*) && path(p1, Company c -/-> Address a) &&
   !path(p2, c -/-> Employee e -/-> a)
```

Figure 4.1: Some examples of PEP

selects field access join points where there is at least one path from `c` to `a` but not through an `Employee` object.

In what follows, the different facets of the path pointcut semantics are going to be discussed in detail.

4.2.2.1 Pattern Matching

Path expression patterns may specify certain objects as source objects, target objects, and intermediate objects of the paths. The associations between objects can be specified by their names. The path pointcut applies pattern matching mechanisms by using the wildcards "`*`" and "`/`" to specify associations between objects along the path.

Each participating object within a path is described in terms of an object pattern, which specifies the runtime type of an object and declares a name for the object that can be used later for the purpose of context exposure. According to AspectJ, it is possible to specify a lexical abstraction over type names using the wildcards "`*`" and "`..`" (cf. [ASPJ (2007); Laddad (2003)]). For instance, the type pattern `A*` matches all types whose names start with an `A`.

The field pattern between an object `o1` and an object `o2` describes the field name over which `o2` must be accessible from `o1`. For example, a path pattern "`o1 -f1-> o2`" describes a selection criterion where object `o2` must be accessible from object `o1` via a field named `f1`. Likewise to type patterns, field patterns also provide lexical abstractions via the wildcard "`*`". So, the path pattern "`o1 -*-> o2`" describes all possible direct relationships between `o1` and `o2`. Direct relationships reflect reference paths from `o1` to `o2` that consist of only one edge. In other words, object `o2` is the value of a given field of object `o1`.

The wildcard "/" is used as a special construct for field patterns. This operator permits to specify indirect relationships between the objects. For example, a path pattern "o1 -/-> o2" describes a selection criterion where object o2 is reachable from object o1 over a path of any length.

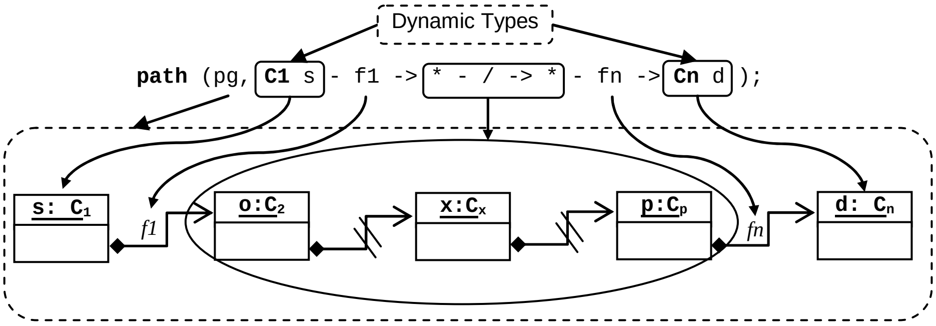


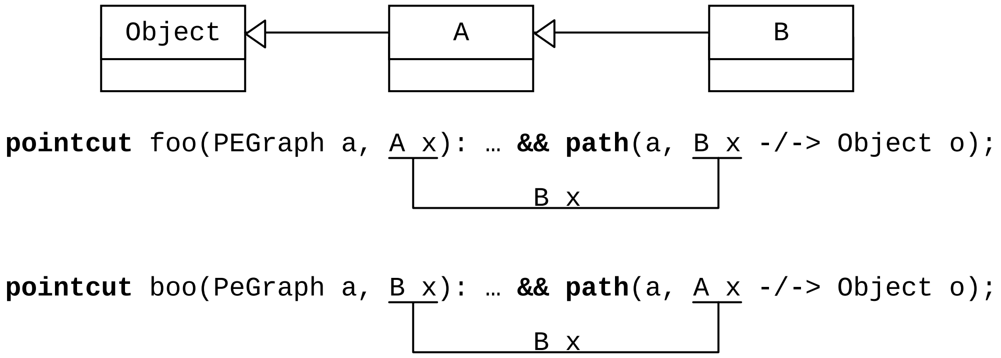
Figure 4.2: An example of PEP

An exemplary path expression pointcut is illustrated in Figure 4.2. The pointcut that uses this PEP selects all join points, if there exists at least one path in the object graph from the source object s of a dynamic type C1 to the destination object d of type Cn. The first edge in each path must be an association between s and the object represented by the field f1 in s, say e.g. o. Similarly, the last edge should be an association between an object that has a field named fn, say e.g. p and the destination object d. The area inside the oval at the center of the path corresponds to any number (≥ 0) of possible inner objects (nodes), which are specified by the wildcard /. The indirect relationship "//" between objects o and x and between objects x to p corresponds to the visualization of *Join Point Designation Diagrams* [Hanenberg et al. (2007); Stein et al. (2004, 2006)]. All object information (objects and their relationships) inside the dashed rounded rectangle defines the PEGraph object pg.

4.2.2.2 Parameter Bindings and Context Exposure in PEP

Parameter bindings and context exposure mechanisms were introduced earlier in Section 2.2.2.6. With respect to the path pointcut, objects described within a path pattern can be directly bound to the corresponding pointcut variables that will be exposed to the aspect.

Context exposure in the presence of path patterns works similar to context exposure in AspectJ. First, all object identifiers being used within a path pattern are bound to the same identifiers declared within the pointcut header. Since a pointcut header already declares a dynamic type for objects described by a dynamic pointcut (corresponding to the definition of `this`, `target` or `args`), PEP also permits to do so: If a pointcut header already specifies a dynamic type on an object declared within a path pattern, it is allowed to leave out the type pattern within the object pattern. In case a type is specified within the header as well as within the path pattern, the actual type is considered to be the most specific subtype given that the mentioned types are in a subtype relationship.



```
pointcut foo(PEGraph a, A x): … && path(a, B x -/-> Object o);

pointcut boo(PeGraph a, B x): … && path(a, A x -/-> Object o);
```

Figure 4.3: Dynamic type resolving in PEP

Figure 4.3 illustrates an example of type resolving in PEP. If within the header (pointcut `foo`) a type `A` for variable `x` is defined and within the path pointcut the type for `x` is declared to be `B` (whereby `B` is a subtype of `A`), the dynamic type of `x` is `B`. Correspondingly, if within the header (pointcut `boo`) a type `B` for variable `x` is defined and within the path pointcut the type for `x` is declared to be `A`, the dynamic type is also `B`.

Figure 4.4 illustrates a situation where a division `d` has one employee `e` whose address is `a1` and one customer `c` that has the same `billTo` and `shipTo` addresses (object `a2`). Consider the following pointcut specification:

```
pointcut p1(PEGraph g, Employee em, Address addr):
  path(g, emp -*-> addr) ...;
```

The only matching path is the one between the objects `e` and `a1` that contains only one edge with label `address`. This will construct the `PEGraph` object `g`, which will be a part of the resulting bindings. The rest of the resulting parameter bindings are the binding of the variables `emp` and `addr` to the objects `e` and `a1`, respectively. These bindings are the relevant join point properties of a potentially selected join point and they are exposed to the pointcut and the associated advice by means of the context exposure mechanism.

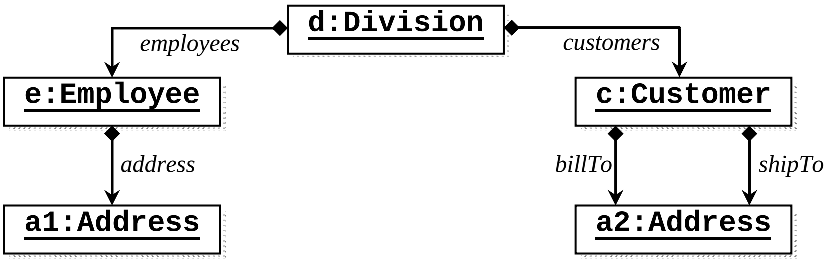


Figure 4.4: Collaboration diagram between divisions, employees and addresses

The associations in the path patterns can be specified by names. As an example, consider the following pointcut specification:

```
pointcut p2(PEGraph g, Customer cust, Address addr):
  path(g, cust -billTo-> addr);
```

The association name `billTo` is specified explicitly, hence, the matching process will return one path: `c -billTo-> a2`. Similar to the previous example, the `PEGraph` object `g` will contain only this path and will be added to the rest of the evaluated binding: `(cust = c, addr = a2)`. This set of parameter binding will be exposed to the aspect.

In many cases, more than one matching path may result in a single valid set of parameter binding. For example, the following path pattern, in pointcut `p3`, matches two paths in the object graph of Figure 4.4: `c -billTo-> a2` and `c -shipTo-> a2`. However, the only possible bindings for the variables `cust` and addr are the objects c and a2, respectively. The exposed context will include these bindings along with the graph object g constructed from the two matching paths. The graph g consists of the objects c and a2 and the two relationships billTo and ShipTo.

```
pointcut p3(PEGraph g, Customer cust, Address addr):
  path(g, cust -*-> addr) ...;
```

Note that this mechanism is about selecting the matching paths as well as exposing them to the aspect context. The PEGraph object g contains all object information related to the selected join point, i.e., objects (c and a2) and their two relationships (billTo and shipTo). Therefore the pointcuts and advice get access easily to all relevant join point context.[1]

According to the semantics of the path expression pointcuts, the created PEGraph object at a given join point depends also on the resolved bindings. That means each distinct parameter binding has its own corresponding PEGraph object, which ensures exposing only relevant information to the join point. The relevant information consists of the objects and their relationships that are included in the matching paths even if these paths contain cycles. Note that in Figure 4.4, objects d, e and a1 along with their referencing field information, i.e. the field names customers, employees, and address, are excluded from the result of the path pointcut since this information is not relevant to the given path pattern.

The presence of cycles in object graphs raises an important question regarding the termination of PEP evaluation. If a cycle appears in a matching path then it must be included in the resulting path graph. To guarantee the termination feature of the path pointcut, one constraint has been added to the algorithm of path pointcut evaluation: cycles should not be traversed more than once except when it is necessary to traverse them again to fulfill the required set of bindings. Notice that the matching algorithm of PEP has to detect the cycles during the traversal of the object graph, and it has to add them to the resulting PEGraph, if they occur in a matching path.

Consider the collaboration diagram in Figure 4.5 and the following pointcut:

[1] This is in contrast to the earlier version of PEP [Al-Mansari and Hanenberg (2006)] with which the exposed context would be the bindings (cust = c, addr = a2), where the aspect lacks the access to the relationship information between c and a2.

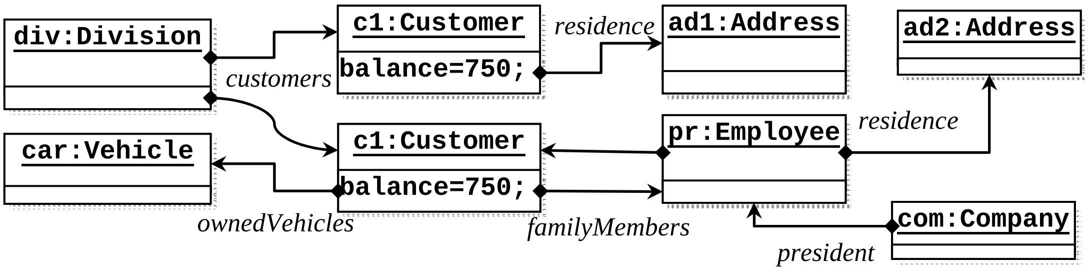
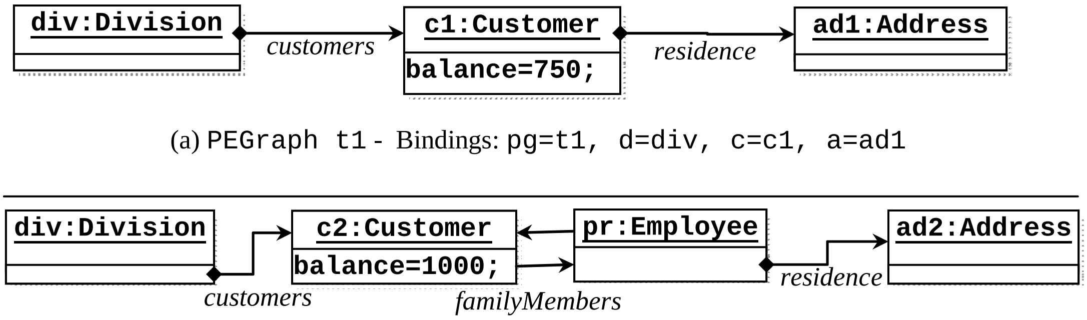


Figure 4.5: Infinite number of paths between objects div and ad2

```
pointcut divToAdd(PEGraph pg, Division d, Customer c, Address a):
  path(pg, d -/-> c -/-> a) ...;
```

According to the given path expression pattern, there is one matching path from `div` to `ad1` via `c1`, which constructs a temporary `PEGraph` object `t1` that is bound to the variable `pg`. The resolved binding is: `(pg=t1, d=div, c=c1, a=ad1)` as shown in Figure 4.6-(a).



(a) PEGraph t1 - Bindings: pg=t1, d=div, c=c1, a=ad1

Figure 4.6: Two PEGraph objects, each for a different set of bindings

On the other hand, there is an infinite number of matching paths from `div` to `ad2` via `c2` due to the presence of the cycle between the objects `pr` and `c2`. Suppose that the traversal algorithm visits `div` then `c2` and finally reaches `pr`, if it chooses to traverse through the edge labeled `familyMembers` then it will visit `c2` again, detect the cycle, save all information (objects and relations), and finally will return back to `pr` and follow the other edge to the object `ad2`. At this point the traversal algorithm finds a matching path as well as valid distinct bindings for d, c and a. The whole path including the cycle will be put in a temporary PEGraph object t2 as in Figure 4.6-(b), and then the pointcut resolves the second binding: (pg=t2, d=div, c=c2, a=ad2). The pointcut evaluation stops afterwards since there are no more matching paths.

Path expression pointcuts support a sort of *unification property* similar to the one in the logic programming [Darlington et al. (1986); Sterling and Shapiro (1994)]. In the scope of a given pointcut, more than one path expression pattern can be used. In addition to the available traditional context exposure pointcut designators such as this, target, and args, the pointcut definition may contain more than one occurrences of a single variable. This is not valid in AspectJ, however, in the presence of path expression pointcuts this is permitted since there is a means to unify all these occurrences to refer to the same corresponding exposed object.

For example, consider the following pointcut:

```
pointcut pc(PEGraph p1, PEGraph p2, Address a):
  set(* *.*) && target(a) && path(p1, Company c -/-> a)
  && !path(p2, c -/-> Employee e -/-> a);
```

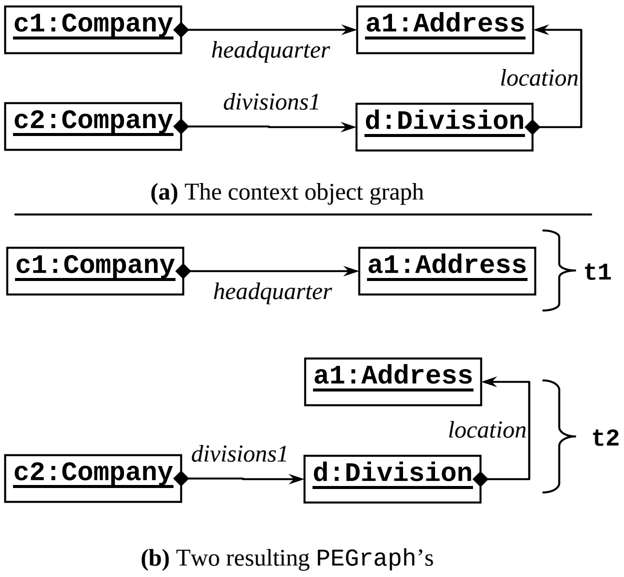
There are three occurrences of the variable name a, twice as a destination in the two given path expressions and it is used in the target pointcut. The path pointcut allows multiple occurrences of the same variable name in one or more path expressions and unifies these occurrences to be bound to the same value.

There are many situations where the developers need to expose the whole matching PEGraph instead of multiple parts of it. Such situations occur when it is required to designate shared objects that are part of the matching PEGraph. PEP supports object-sharing relationships. This depends on the formal parameters of the enclosing pointcut. For example, consider the following pointcut:

```
pointcut shareLost(PEGraph g, Company c, Address a):
  ... && path(g, c -/-> a);
```

If this pointcut applies on the object graph in part(a) of Figure 4.7, then, the result consists of two distinct set of parameter bindings: (g=t1, c=c1, a=a1) and (g=t2, c=c2, a=a1) as shown in part (b) of the figure.

In order to capture the shared object a1, one can specify the following pointcut:



(a) The context object graph

(b) Two resulting PEGraph's

Figure 4.7: A shared `Address` object and two resulting `PEGraph` from pointcut `shareLost`

```
pointcut shareMaintained(PEGraph g, Address a):
  ... && path(g, c -/-> a);
```

Since there is only one distinct binding for `a1`, there is only one resulting `PEGraph` which is the same as the original object graph. The exposed binding is `(g, a=a1)`.

4.2.2.3 PEP as an Expressive Pointcut

Path expressions are proved to be expressive enough to challenge any updates to the underlying code or its structure. An example of that is the traversal strategies, which are used in DemeterJ, a language for adaptive programming [Lieberherr and Orleans (1997); Orleans and Lieberherr (2001)].

This is the same case for the path expression pointcuts. For example, assume that it is required to select any change to an address object that belongs to division objects of each company. Then, PEP can be used to get access to the changed address object along with its owner division and company objects. This functionality is given by the following pointcut definition:

```
pointcut addChg(PEGraph g, Company c, Division d, Address a):
  path(g, c -/-> d -/-> a) && set(* Address.*) && target(a);
```

Suppose that the company model given in Section 3.1 in the last chapter needs to be modified such that each division is divided into a number of departments in which the employees are grouped. Figure 4.8 depicts the part of the object model that illustrates the changes. Despite of these changes, the PEP in `addChg` pointcut is up-to-date and does not require any changes since it is still reflecting the relationship between company, division and address objects.

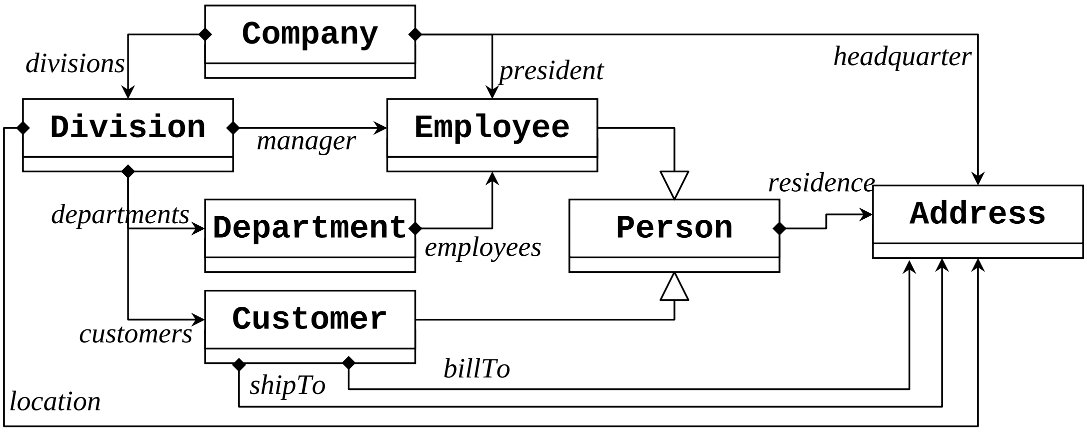


Figure 4.8: Company object model modified to have division's departments

4.2.2.4 Advice Execution Mechanism

One consequence of path expressions is that it is possible (or rather the usual case) that a pointcut containing a path expression provides a number of different results as valid parameter bindings in the pointcut. Because of that, a new semantics for advice execution is necessary that differs widely from the semantics of advice execution such as in AspectJ. The associated advice to a given pointcut must run at the selected join point as many times as the number of available valid parameter bindings. Such an approach of advice execution has been already proposed in other pointcut languages (cf. e.g. [Sakurai et al. (2004)]).

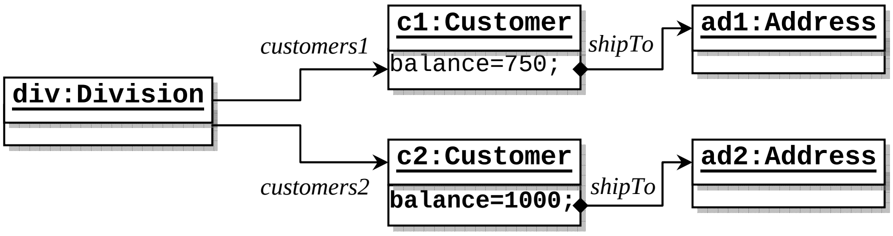


Figure 4.9: Two different paths from a division to an address resolve to two bindings

For example, the object graph of Figure 4.9 consists of a `Division` object that is related to two customer objects ad their `shipTo` addresses. Note how the field `customers` of the class `Division` is represented in the figure above by two different relations, `customers1` and `customers2`. This representation of collections and arrays will be discussed later on in this chapter. Consider the path pattern in the following path pointcut:

```
path(g, Division d -*-> Customer c -*-> Address a)
```

It matches two paths in the object graph of Figure 4.9 with two different bindings. The first path is `div -customers1-> c1 -shipTo-> ad1` and corresponds to the bindings set `(d=div, c=c1, a=ad1)`, hence, this path constructs the path graph object `g` that will be added to the first binding set. The second path is `div -customers2-> c2 -shipTo-> ad2` and corresponds to the binding set `(d=div, c=c2, a=ad2)` in addition to binding the variable `g` to the `PEGraph` object constructed from the second matching path. Hence, each single advice that is associated with this pointcut must be executed two times for each single binding.

Assume that there are two concurrent transactions, each updates one of the objects `ad1` and `ad2` and that the developer wants to run these concurrent changes in a descending order according to the value of the customer's `balance` field. Since the balance of `c2` is greater than the one of `c1`, the advice must be executed first with the binding: `(g, d=div, c=c2, a=ad2)`.

Therefore, an explicit ordering of multiple advice executions might become important and the developers require means to specify this ordering.

4.2.2.5 Ordering Multiple Advice Executions

Multiple advice executions that are resulted from a given path pointcut at the same join point can be ordered by associating the path pointcut with an extra construct called `orderBy`. It takes one parameter representing the name of the method that contains the ordering code specified by the developer. In case the developer is not interested in ordering these advice executions, no `orderBy` clause must be specified.

The ordering method is similar to the `compare` method of the `Comparable` interface in the Java API. The ordering clause maintains a list of all valid parameter binding sets. It uses the ordering method rules specified by the developer to produce an ordered list of those parameter binding sets. The return type of the ordering method is `boolean`. This return value is a result of a comparison between an object from the first parameter and a corresponding object with the same index from the second parameter.

Note that a single ordering method can be used by several pointcuts given that these pointcuts have the same parameters in the same order.

One may argue about why not to provide an explicit construct such as `declare precedence` that is available in AspectJ. However, there are a number of reasons behind this design decision.

First, it gives developers the ability to specify their own ordering rules based on the resulting parameter bindings in a fine grained manner. This is due to that the competing executions take place on the same advice but not between different pieces of advice from different aspects. Second, each pointcut that includes a path pointcut may require its own ordering criteria. Third, within the associated ordering method, the developer can use the exposed objects to reach any other referenced object that she/he wants to use for the ordering.

The parameters of the ordering method could be generally given by an array of the general supertype `Object`. The elements are in the order given by the pointcut or advice parameters. The following code illustrates a concrete example of the `orderBy` clause usage. The anonymous pointcut of the `before` advice selects the field set join points where the target is an `Address` object `a` and where at least one path from a `Division` object `d` to `a` via `Customer` objects exists.

According to the object graph of Figure 4.9 above, there are two resulting sets of parameter bindings as mentioned in the previous example: (g, c=c1, a=ad1) and (g, c=c2, a=ad2).

```
public boolean addrChg(Object[] o1, Object[] o2) {
  Customer cust1 = (Customer) o1[1];
  Customer cust2 = (Customer) o2[1];
  return cust1.getBalance() > cust2.getBalance();
}
before(PEGraph g, Customer c, Address a):
  set(* *) && target(a) &&
  path(g, Division d -/-> c -/-> a) orderBy(this.addrChg) { //... }
```

The parameter of the orderBy is the name of the method addrChg that has two array parameters each representing a single binding set. Since the ordering is chosen to be done according to the Customer's field balance, the method extracts the second element from each array and casts them to Customer. The method returns the comparison result between their balance fields. In this example, the second customer object, i.e. c2, has a greater balance value than the first one, which means that the before advice will run first on the second binding set (g, c=c2, a=ad2). Note that the Division object is not part of the bindings because there is no corresponding parameter in the advice header.

4.3 Discussion

In the rest of this chapter, some issues regarding the above description of PEP are discussed.

4.3.1 PEP Comprehension

In general, aspects are a very powerful and complex language mechanism, combining features of both dynamic scoping and continuation manipulation. Much of the research on aspect-oriented programming focused on applying aspects in various problem domains. Another part of AOP research focused on extending the current AO systems and tools to help solving those problems. Adding new constructs to existing AOP languages, on the other hand, is a very complex and tedious task.

From the programming language designer perspective, it is necessary to have a precise and elegant formalism of the existing languages as well as the new constructs. Informal description of the new features is less comprehensive and suffers from ambiguity, whereas a formal specification eases the comprehension of the new features and resolves any non-intentional ambiguities.

Path expression pointcuts are no exception. The informal description of the path pointcut in the last section provides a good guide to how path pointcuts can be applied and used. However, from a programming language designer point of view, this description is not enough to clarify how the introduced concepts and components of the path pointcut relates to each other and how they can be implemented. This ambiguous description complicates the task of the designer when introducing path pointcut to an existing aspect-oriented programming language.

In order to illustrate the above argument, consider the object graph in Figure 4.10. The persistent address instance `a1` is being changed by the object `pcc`, these two objects represent the local context to the object change join point (inside the dashed area). If a cascading-version locking policy is performed on the objects of the persisting container `p1`, the version fields of the object `a1` and its non-local owner objects, `s1`, `e`, `d`, and `c2` must be checked and updated when the changes on `a1` are committed.

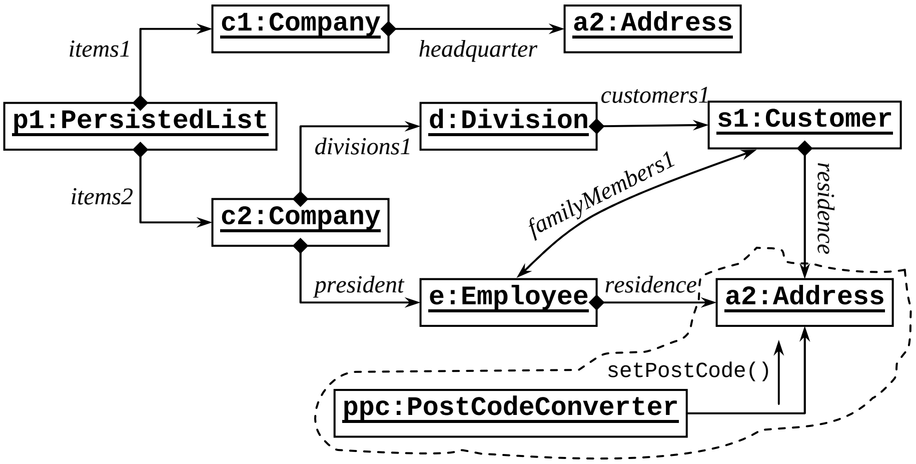


Figure 4.10: The local join point properties are inside the dashed area

In terms of aspects, one needs to select each field change join point on any object that belongs to a persistent list. Besides, the aspect must get access to all non-local owner objects of the dirty object. The situation is not about a reachability problem only, rather, it is also more about accessing specific parts of the object graph. Hence, the situation is a good practice for the path pointcut instead of bothering the developer with complex workarounds in order to get access to the non-local object information. The following pointcut provides the needed functionality:

```
pointcut pc(PEGraph pg, PersistedList p,Object o):
  set(* *) && target(o) && path(pg, p -/-> o) …;
```

Now, in order to describe the meaning of this path pointcut in terms of the natural English language, one can write the following:

> ”The result of this path pointcut evaluation consists of the matching paths in the current object graph to the path expression pattern `p -/-> o`. Those paths are of any length and are starting from a `PersistedList` object and ending at any object referenced from this `PersistedList` object. Both variables `p` and `o` are to be bound to the corresponding *start* and *end* objects of the matching path, respectively. All matching paths construct the `PEGraph` object that is to be bound to the variable `pg` such that a single `pg` instance contains the needed bound objects for `p` and `o`. The number of the exposed context from this pointcut equals the number of distinct valid bindings, here it is one.”

The attempt to make this statement precise has rendered it almost hard to comprehend. The programming language designer is barely able to figure out a suitable representation for each component, e.g. the path or the binding. The statement also does not show how the cycle between the customer `s1` and the employee `e` is represented in the matching paths. Furthermore, the statement is ambiguous with respect to the description of the relations between the matching paths, the bindings, and the resolved actual types of the objects. For example, the address object `a1` is shared between an infinite number of matching paths, so it is not clear how these paths can be grouped together to construct common `PEGraph` objects. As another example, it fails to clarify the meaning of the needed binding's covering.

Basically, the rationale behind this difficulty is that the `PEGraph` as well as the parameter binding are too complex mechanisms that have to be well understood, e.g. in order to be implemented. Hence, informal descriptions of a single or even more than one example cannot be used as a general specification to implement path pointcuts.

Hence, providing a formal semantics to the PEP's can solve this ambiguity and improve the understandability of the concept.

4.3.2 Typing Issues

As stated above, the result from the path pointcut is a subgraph of the whole object graph that contains only the object information relevant to the selected join point. This resulting graph will be assigned to an object of type `PEGraph`. The intention of having such data structure is to make the relevant object information to the selected join point available for the advice so that the developers can access this information easily. Accordingly, it is necessary to provide a suitable and easy to use interface for this graph.

A simple one can be generally specified as follows, where all type information is neglected and all objects will be of the general supertype `Object`:

```
class PEGraph {
  Object start; // the start object
  Object end;   // the end object
  List inners;  // the list of inner objects
  List edges;   // the list of the edges
   // other members such as traversal methods
}
```

For example, consider the situation where each company division provides service for the customers that live within certain post code areas. Then, changing the address of a customer requires to check whether the new address is still inside the area covered by its owner division, otherwise the customer profile has to be moved to another division. Figure 4.11 illustrates a join point where the address of the customer `s1` is being changed by the invocation of `setAddress` method.

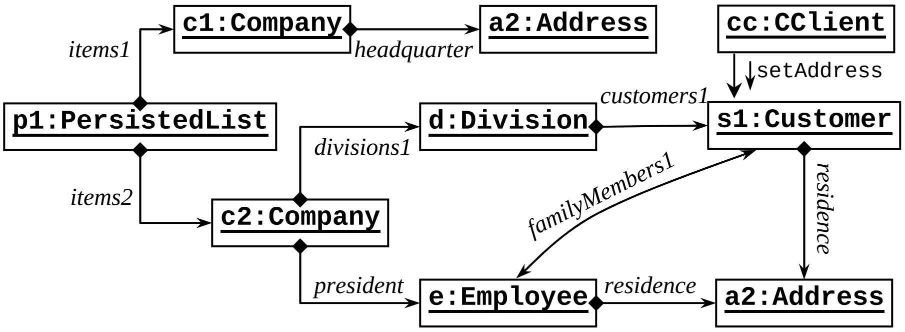


Figure 4.11: The `address` of `Customer s1`'s field is being changed

In terms of aspects, this method execution join point is selected if the updated customer is part of a given persistent list. This reachability information is considered as non-local to the join point. Moreover, in order to adapt this join point, the advice must access the non-local owner `Division` object to check whether the new address is inside the area that is covered by this division. Such information can be provided with the help of the path pointcut as follows:

```
pointcut cusAddChg(PEGraph pg): execution(* Customer.setAddress(*))
  && path(pg, PersistedList p -/-> Division d -*-> Customer s);
```

The resulting `pg` is shown in Figure 4.12. The following advice is responsible for performing the required adaptation.

```
after(PEGraph pg): cusAddChg(pg) {
  // get the owner d from pg.inners
  for(Object o:pg.inners) {
    if(o instanceof Division) {
      Division d = (Division) o;
      Customer s = (Customer) pg.end;
      // code to check if the new address of s
      // is in the service area of d ...
    }
  }
}
```

The adaptation of the selected join point requires manipulating the division object `d` as well as the customer object `s1`. However, we cannot benefit from the type information given by the pattern to get the right division and customer objects unless we have included them in the pointcut header as bound variables.

The only way is to traverse `pg.inners` to get the objects where we cannot avoid using type-cast.

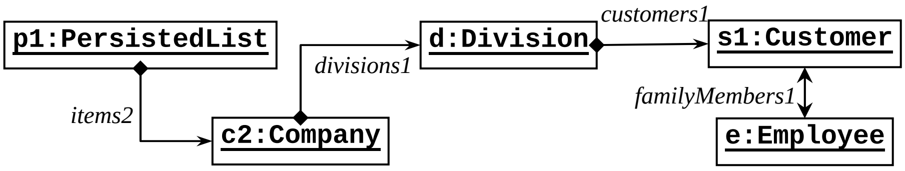


Figure 4.12: The resulting `PEGraph` from pointcut `cusAddChg`

As a consequence, providing a suitable type for `PEGraph` becomes necessary in order for the advice to obtain the already known type information.

On the other hand, there is another typing problem in the above description of PEP. The developers have to type-cast the bound objects inside the ordering method. This again breaks the type-safety of PEP. Hence, a proper type for the ordering method is required.

Finally, any solution for these typing problems requires ensuring the correct derivation of these types. Therefore, a type system becomes a must.

4.4 Chapter Summary

This chapter presented the informal description of the path expression pointcut by using plenty of examples. It covers many concepts that are part of this rich and complex construct. Some issues were parts and properties of the path expressions itself. Other concepts were a result of applying path expressions in aspect-oriented programming.

The chapter started by defining the concrete syntax of PEP and its support of pattern matching for the names of the types and associations. As a binder pointcut, PEP provides its own mechanism of parameter bindings and context exposure which is discussed in this chapter. Also, the chapter described the extension to the advice execution mechanism in aspect-oriented programming. This extension was necessary due to that PEP could produce multiple distinct valid bindings for which the advice must be executed more than once at a selected join point. Accordingly, this chapter described how to order these executions.

The chapter ends by motivating the need for formal specifications for the PEP. It also motivates the need for suitable types of the path expression graphs for the exposure mechanism and the binding lists for the ordering schema. This means that there is also a need to ensure the correctness of these types inside the aspect, which can be achieved by a proper type system.

Chapter 5

Formal Semantics and Type System

Formal specifications are globally considered to be efficient means to gain a clear understanding of programming languages and constructs. Such an unambiguous understanding is required for the correct implementation of the programming language as well as for its usage. Path expression pointcuts are not an exception, especially, as illustrated in the last chapter, because they gather a number of complex features such as object graphs, bindings and `PEGraph`.

This chapter presents a *denotational formal semantics* for PEP that can work as a tool to understand PEP without any ambiguities. There are two main reasons for choosing denotational semantics. First, this technique is considered a good approach to describe the meaning of different computer science concepts, including path expressions themselves, e.g. [Draper et al. (2007); Wadler (2000)]. Second, path pointcuts are applied to object graphs, which in turn can be modeled easily in terms of the theory of sets. This mathematical model works as a basis for the denotational semantics presented here.

Then, following the argument presented at the end of the last chapter, a proper type for `PEGraph` is proposed along with a simple type system for the path pointcut.

This chapter is organized as follows. Section 5.1 gives a short introduction about the denotational semantics. Section 5.2 presents the mathematical model of the object graphs that works as the base of the path pointcut formalization.

The syntactic specification of the construct will then be presented in Section 5.3. Section 5.4 presents the denotational semantics of the path expression pointcut. The integration of this formalization with current semantics of pointcut languages is discussed in Section 5.6. Section 5.7 presents a parameterized type for `PEGraph` objects, then proposes a type system for the path pointcut. A short summary of the chapter is given in Section 5.8.

5.1 Denotational Semantics

Denotational semantics allows to specify the meaning of the programs by mapping them to abstract but precise mathematical concepts and domains such as $Integers$, $Boolean$ values and functions. The idea of denotational semantics is to associate an appropriate denotational mathematical object with each phrase of the language.

The process of defining a denotational semantics runs through two main phases: *Syntactic world* and *semantics world.* Traditionally, special brackets, the *emphatic brackets* $[\![\,]\!]$ are used to separate the syntactic world from the semantic world. If p is a syntactic phrase in a programming language, a denotational specification of the language will define a mapping meaning, so that meaning $[\![p]\!]$ is the denotation of p; namely, an abstract mathematical entity that models the semantics of p.

In the syntactic world, one should specify first the syntactic domains of the syntactic objects that may occur in the phrases of the language, e.g., $Numeral$, $Variable$, and $Expression$. As a convention, corresponding metavariables for these domains are used:

$$\begin{array}{lcl} N & : & Numeral \\ V & : & Variable \\ E & : & Expression \end{array}$$

Next, the abstract syntax of the language must be specified based on its concrete syntax and using the syntactic domains. For example, if the concrete syntax of a simple calculator is as follows:

$$
\begin{array}{lcl}
\mathit{Expression} & ::= & \mathit{Expression} + \mathit{Term} \\
 & | & \mathit{Expression} - \mathit{Term} \\
 & | & \mathit{Term} \\
\mathit{Term} & ::= & \mathit{Term} * \mathit{Primary} \\
 & | & \mathit{Term} \,/\, \mathit{Primary} \\
 & | & \mathit{Primary} \\
\mathit{Primary} & ::= & \mathit{Numeral} \\
 & | & (\mathit{Expression}) \\
 & | & \texttt{let}\ \mathit{Variable} \ = \ \mathit{Expression}\ \texttt{in}\ \mathit{Expression} \\
 & | & \mathit{Variable}
\end{array}
$$

The abstract syntax that is matching directly the abstract syntax tree could be specified as follows:

$$
\begin{array}{lcl}
E & ::= & E + E \\
 & | & E - E \\
 & | & E * E \\
 & | & E \,/\, E \\
 & | & N \\
 & | & (\,E\,) \\
 & | & \texttt{let}\ V \ = \ E\,\texttt{in}\,E \\
 & | & V
\end{array}
$$

In the semantic world, the semantic domains are *sets* of mathematical objects of a particular form. For example, the following are semantic domains:

$$
\begin{array}{lcl}
\mathit{Integer} & = & \{\ldots,\, -2,\, -2,\, 0,\, 1,\, 2,\, \ldots\} \\
\mathit{Store} & = & (\mathit{Variable} \times \mathit{Integer})
\end{array}
$$

The domain *Store* consists of sets of bindings (mapping variable names to integer values). The domain *Integer* is used only to represent the *Numeral* syntactic domain since the division operation "/" is considered to produce integer values only. The notation $A \longrightarrow B$ is used to denote the set of functions with domain A and codomain B.

The *Cartesian* product between two sets also specifies a semantic domain. So, $A \times B$ denotes the set of tuples of the form $(a,\, b)$, where $a \in A$ and $b \in B$. This set can work as a semantic domain or codomain.

The connection between the syntax and the semantics world is achieved by means of the semantic functions. These functions map objects of the syntactic world into objects in the semantic world. For example,

$$eval \;:\; Expression \times Store \longrightarrow Integer$$

The function *eval* maps the syntactic expressions to the semantic values that are integers using some auxiliary functions that represent the corresponding mathematical operations. These auxiliary semantic functions are:

$$\begin{array}{lcl} add & : & Integer \times Integer \longrightarrow Integer \\ add(N_1,\, N_2) & = & N_1 + N_2 \end{array}$$

$$\begin{array}{lcl} mul & : & Integer \times Integer \longrightarrow Integer \\ mul(N_1,\, N_2) & = & N_1 * N_2 \end{array}$$

$$\begin{array}{lcl} sub & : & Integer \times Integer \longrightarrow Integer \\ sub(N_1,\, N_2) & = & N_1 - N_2 \end{array}$$

$$\begin{array}{lcl} div & : & Integer \times Integer \longrightarrow Integer \\ div(N_1,\, N_2) & = & N_1 \,/\, N_2 \end{array}$$

Regarding the operations on the *Store*, two more auxiliary functions are needed: The function *value* that returns the integer value of a given variable and the function *extend* that allocates a place in *Store* to store the value of a new variable after performing the `let-in` expression. Let δ ranges over *Store* in the rest of this example, then the functions *value* and *extends* are defined as follows:

$$\begin{array}{lcl} value & : & Variable \times Store \longrightarrow Integer \\ value(V,\, \delta) & = & N, \text{ where } (V,\, N) \in \delta \end{array}$$

$$\begin{array}{lcl} extend & : & Variable \times Integer \times Store \longrightarrow Store \\ extend(V,\, N,\, \delta) & = & \delta', \text{ where } value(V,\, \delta') = N \end{array}$$

Finally the semantic equations are defined to illustrate how the functions manipulate each pattern from the syntactic world. For example,

$$eval[\![E_1 + E_2]\!]_\delta \;=\; add(eval[\![E_1]\!]_\delta,\, eval[\![E_2]\!]_\delta)$$

As this example shows, and in fact in many cases, the semantic equation is associated with a context where it should be computed. For example, here $eval[\![E_1 + E_2]\!]_\delta$, depends on the storage that contains the values of E_1 and E_2. The final semantic equations of the calculator example are:

$$\begin{array}{lcl}
eval[\![E_1\ +\ E_2]\!]_\delta & = & add(eval[\![E_1]\!]_\delta,\ eval[\![E_2]\!]_\delta) \\
eval[\![E_1\ -\ E_2]\!]_\delta & = & sub(eval[\![E_1]\!]_\delta,\ eval[\![E_2]\!]_\delta) \\
eval[\![E_1\ *\ E_2]\!]_\delta & = & mul(eval[\![E_1]\!]_\delta,\ eval[\![E_2]\!]_\delta) \\
eval[\![E_1\,/\,E_2]\!]_\delta & = & div(eval[\![E_1]\!]_\delta,\ eval[\![E_2]\!]_\delta) \\
eval[\![N]\!]_\delta & = & N \\
eval[\![(E)]\!]_\delta & = & eval[\![E]\!]_\delta \\
eval[\![\texttt{let}\,V = E_1\,\texttt{in}\,E_2]\!]_\delta & = & eval[\![E_2]\!]_{(extend(V,\,eval[\![E_1]\!]_\delta,\delta))} \\
eval[\![V]\!]_\delta & = & value(V,\ \delta)
\end{array}$$

All these equations are simple except $eval[\![\texttt{let}\,V = E_1\,\texttt{in}\,E_2]\!]_\delta$, which means that expression E_1 is evaluated first with respect to the current state of the store δ. Then, the result of this evaluation is assigned to the variable V. This mapping is then added to δ to produce a new state of the store, i.e. δ'. Finally, expression E_2 is evaluated with respect to δ'.

These semantics equations can be used directly to describe the evaluation of any valid expression of the calculator example. For example, assume that the current state of the storage δ is as follows:

$$\begin{array}{lcl}
\nu & = & \{a,\ b\}\,\text{where}\,\nu \in Variable \\
\delta & = & \{(a,\ 5),\ (b,\ 3)\}
\end{array}$$

Then, the following shows two possible applications of the above semantics:

$$\begin{array}{lcl}
eval[\![a + b]\!]_\delta & = & add(eval[\![a]\!]_\delta,\ eval[\![b]\!]_\delta) \\
 & = & add(value(a, \delta),\ value(b,\ \delta)) \\
 & = & add(5,\ 3)\ =\ 5 + 3\ =\ 8
\end{array}$$

$$\begin{array}{lcl}
eval[\![\texttt{let}\,c = b\,\texttt{in}\,a - c]\!]_\delta & = & eval[\![a - c]\!]_{(extend(c,\,eval[\![b]\!]_\delta,\,\delta))} \\
 & = & eval[\![a - c]\!]_{(extend(c,\,value(b,\delta),\,\delta))} \\
 & = & eval[\![a - c]\!]_{(extend(c,\,3,\,\delta))} \\
 & = & eval[\![a - c]\!]_{\delta'},\ \text{where}\,\delta' = \{(a,\ 5),\ (b,\ 3),\ (c,\ 3)\} \\
 & = & sub(eval[\![a]\!]_{\delta'},\ eval[\![c]\!]_{\delta'}) \\
 & = & sub(value(a,\ \delta'),\ value(c,\ \delta')) \\
 & = & sub(5,\ 3)\ =\ 5 - 3\ =\ 2
\end{array}$$

5.2 Formal Mathematical Base Model

In this section, a mathematical model for the object graphs is developed. This model is considered as a basis of the denotational semantics. Unlike to, e.g., the XML documents data models in [Draper et al. (2007); Wadler (2000); Wood

(1998)], the model presented here is defined for directed graphs that mostly contain cycles rather than trees. Hence, it needs to be different because trees do not contain cycles.

This model is applied to the current object graph at a certain point of time, and it is not concerned with changes in the object graph. I.e. it is not the intention to explain via this model how a programming language transfers one object graph into a following one.

5.2.1 The Object Graph Model

The object graph, denoted as $G = (V,\ E,\ T,\ L, atype, types)$, is a directed labeled graph where V is a set of nodes, each represents an object that is associated with a type $t \in T$, the set of types. E is a set of edges. Each edge represents a reference relationship between two given objects. L is a set of labels that represent field names. The set of edges E is defined in the domain $V \times L \times V$. For example, if $e = (u,\ l,\ v) \in E$, there is an edge from the source $u \in V$ to the destination $v \in V$ which is labeled by $l \in L$. In other words, there is a direct relationship between the object u and the object v by the field named l. Moreover, $\nexists\, e_1,\ e_2 \in E\,$s.t.$\,e_1 = (u,\ l,\ v1)$ and $e_2 = (u,\ l,\ v2)$, which ensures that a field has only one value.

Indirect relationships are represented by a sequence of at least two edges. For example, if $e_1 = (u_1,\ l_1,\ u_2)$ and $e_2 = (u_2,\ l_2,\ u_3)$ are elements of E. This indicates an existence of an indirect relationship between the objects represented by the nodes u_1 and u_3 via the node u_2. Such relationships in the object graphs are also known as *reachability* between objects; i.e. object u_3 is reachable from object u_1 since there is a path from u_1 to u_3 via u_2. The paths will be discussed later in this section.

For object-oriented languages, there is a need to model the empty object called `null`, e.g. in Java or `nil` in Smalltalk. This empty object is modeled as a special node called $null \in V$. When an object o references the $null$ object, there is an edge in the object graph between the node that represents o and the $null$ node. The $null$ node may have parent nodes, but it has no children nodes. These properties are described as follows:

$$\nexists\, x \in V,\, l \in L\, s.t.\, (null,\, l,\, x) \in E \tag{5.1}$$

Programming languages such as Java or C++ also provide special language constructs for *Arrays* that permit to access a certain field with at least one index. In this thesis, the array elements are represented as ordinary fields with special labels (starting with the array field name, followed by a natural number corresponding to the array element index). Hence, there is no guarantee that the arrays are ordered data structures when parsing the graph model. Although this could be considered as a weakness of this mathematical model, it is sufficient here, because PEP's do not depend on the ordering of relationships.

Since collections in programming languages such as Java can be implemented using arrays, the above mentioned representation of arrays is sufficient to represent any kind of collections.

Finally, the graph G has three functions *atype*, *types* and *label*. The function *atype* maps each object (node) to its actual type, while the function *types* is a mapping between a type and its supertypes. The function *label* returns the label of a given node.

$$\begin{aligned}
atype &: V \longrightarrow T \\
atype(x) &: t,\ \text{where}\, t \in T \qquad (5.2) \\
types &: T \longrightarrow Set(T) \\
types(t) &: \{t,\, \texttt{Object}\} \cup \{t_1 \in T |\, t_1\, \text{is a supertype of}\ t\} \qquad (5.3) \\
label &: E \longrightarrow String \\
label(e) &: l,\ \text{where}\, e = (u,\, l,\, v) \in E \qquad (5.4)
\end{aligned}$$

Note that `Object`, here, means the root supertype of all types in object-oriented programming languages such as Java and Smalltalk. Moreover, since the subtyping relationship is reflexive [Cardelli (1997)], a given type is a supertype of itself.

The end nodes of an edge $e \in E$ can be extracted by using the auxiliary functions *source* and *target*:

$$\begin{aligned} source &: E \longrightarrow N \\ source(e) &: u, \text{ where } e = (u,\, l,\, v) \in E \qquad (5.5) \\ target &: E \longrightarrow N \\ target(e) &: v, \text{ where } e = (u,\, l,\, v) \in E \qquad (5.6) \end{aligned}$$

5.2.2 An Example of the Object Graph Model

The model of the object graphs that is specified above does not consider the way that object graphs are constructed. This is going to be explained later on in this chapter when the complete aspect language is presented. For the moment, it is assumed that a given collaboration diagram is enough to show the different components of the corresponding object graph G.

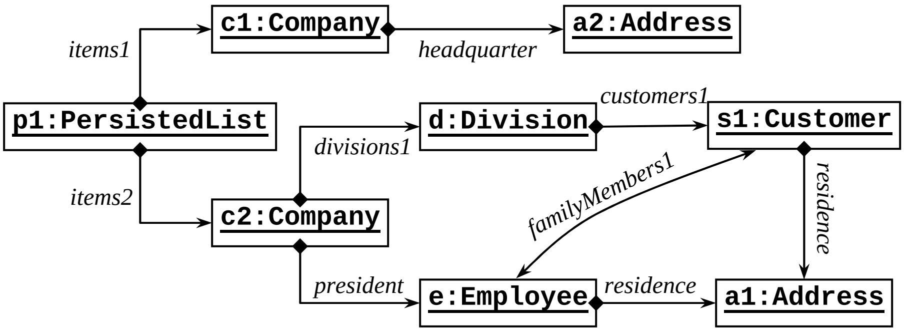


Figure 5.1: Example of an object graph

Figure 5.1 depicts the collaboration diagram that is taken from the collaboration diagram in Figure 4.10 that is shown in the last chapter. According to this diagram, the sets V, E, L and T of the object graph G are given as follows:

$$V = \{p1,\ c1,\ c2,\ d,\ e,\ s1,\ a1,\ a2\}$$

$$\begin{aligned} E = \{&(p1,\ items1,\ c1),\ (p1,\ items2,\ c2),\ (c1,\ headquarter,\ a2),\ (c2,\ divisions1,\ d), \\ &(d,\ customers1,\ s1),\ (s1,\ residence,\ a1),\ (c2,\ president,\ e), \\ &(e,\ familyMembers1,\ s1),\ (s1,\ familyMembers1,\ e),\ (e,\ residence,\ a1)\ \} \end{aligned}$$

$$L = \{ \; items1,\; items2,\; divisions1,\; president,\; headquarter,\; customers1,\; familyMembers1,\; residence \; \}$$

$$T = \{PersistedList,\; Division,\; Company,\; Employee,\; Customer,\; Address\}$$

Notice how the `items` field that is of type `List` is modeled. `PersistedList` contains the two company objects `c1` and `c2`, so there exist two edges from it to these objects via labels `items1` and `items2`, respectively.

The specification of functions *label*, *atype* and *types* that are part of G is given below as sets of mappings from the domain to the codomain of each function:

$$\begin{aligned} label \; = \{ \; & (p1,\; items1,\; c1) \to items1,\; (s1,\; familyMembers1,\; e) \to familyMembers1,\\ & (c1,\; headquarter,\; a2) \to headquarter,\; (c2,\; divisions1,\; d) \to divisions1,\\ & (d,\; customers1,\; s1) \to customers1,\; (s1,\; residence,\; a1) \to residence,\\ & (p1,\; items2,\; c2) \to items2,\; (e,\; familyMembers1,\; s1) \to familyMembers1,\\ & (c2,\; president,\; e) \to president,\; (e,\; residence,\; a1) \to residence \; \} \end{aligned}$$

$$\begin{aligned} atype \; = \{ & p1 \to PersistedList,\; c1 \to Company,\; c2 \to Company,\; d \to Division,\\ & e \to Employee,\; s1 \to Customer,\; a1 \to Address,\; a2 \to Address \; \} \end{aligned}$$

$$\begin{aligned} types \; = \{ \; & PersistedList \to \{PersistedList,\; Object\},\; Address \to \{Address,\; Object\},\\ & Division \to \{Division,\; Object\},\; Company \to \{Company,\; Object\},\\ & Customer \to \{Customer,\; Person,\; Object\},\\ & Employee \to \{Employee,\; Person,\; Object\} \; \} \end{aligned}$$

Finally, here are two examples of the usage of the two functions *source* and *target*:

$$source(c1,\; headquarter,\; a2) \; = \; c1$$

$$target(c1,\; headquarter,\; a2) \; = \; a2$$

5.3 The Syntactic World

This section is about specifying the syntactic domains and consequently defining the abstract syntax of the path expression pointcut.

In order to simplify the definition of the denotational semantics of PEP, an abstract syntax of the construct must be specified. The abstract syntax is defined as a set of BNF rules and is used to communicate the structure of phrases in terms of their semantics in a programming language as trees. These trees are called *abstract syntax trees* (cf. [Aho et al. (1986)]). The abstract syntax tree captures the syntactic structure of any expression in the programming language completely in a much simpler form. The content of the abstract syntax tree is specified by a collection of syntactic categories or domains and the abstract syntax that tells how categories are decomposed into other categories or tokens.

The abstract syntax of PEP is derived from the following concrete syntax of the path pointcut:

PATHEXPRESSIONPOINTCUT SYNTAX

$$\begin{array}{lcl}
PEP & ::= & \text{“}\texttt{path}\text{”}\,\text{“(”}\ PGT\ id\ \text{“,”}\ PathPattern\ \text{“)”} \\
PathPattern & ::= & (ObjPattern\ \text{“}-\text{”}FieldPattern\text{“}\rightarrow\text{”})^{+}\ ObjPattern \\
ObjPattern & ::= & TypePattern\ id \mid id \\
FieldPattern & ::= & IdPattern \mid \text{“/”} \mid \text{“}*\text{”} \\
IdPattern & ::= & \text{“}*\text{”}\ [IdPattern] \mid id\ \text{“}*\text{”} \mid id \\
TypePattern & ::= & Defined\ according\ to\ the\ AspectJ\ syntax \\
Id & ::= & Defined\ according\ to\ the\ Java\ syntax \\
PGT & ::= & \texttt{PEGraph} \mid \texttt{PEGraph<T, PEGraph>}
\end{array}$$

First, the abstract syntactic domains and their metavariables are specified:

$$\begin{array}{rcl}
PEP & : & PathExpressionPointcut \\
P & : & PathPattern \\
F & : & FieldPattern \\
O & : & ObjectPattern \\
I & : & Identifier \\
R & : & Relation \\
T & : & Type \\
PGT & : & PEGraphType
\end{array} \tag{5.7}$$

For the purpose of simplicity, the following text will assume that *IdPattern* and *TypePattern* are simple identifiers as defined in languages such as Java: An identifier is an unlimited-length sequence of Java letters and Java digits, the first of which must be a Java letter [Gosling et al. (1996)]. This is, in fact, the representation of both in the abstract syntax tree.[1]

The intention of the denotational semantics is to describe the meaning of the language. Hence, the notion of types in this context does not mean that it is necessary to provide a type judgment for types in the path pointcuts. This will be discussed in a later section of this chapter. Here, *Type* and *PEGraphType* are simple type identifiers.

As a convention, the metavariables are used in the specification of the abstract syntax of the path expression pointcut as follows:

$$\begin{array}{rcl} PEP & ::= & \texttt{path}(PGT\ I,\ P) \\ P & ::= & R^{+}\ O \\ R & ::= & O\ F \\ O & ::= & T\ I \,|\, I \\ F & ::= & I \,|\, / \,|\, * \\ T & ::= & I \\ PGT & ::= & I \end{array} \tag{5.8}$$

Note that the use of the postfix "+" for the relation in the first production rule can be substituted by a repeated non-terminal R in its production rule. Both cases indicate that the pattern must contain at least one relationship. Furthermore, the "$*$" in the field pattern describes the terminal symbol "$*$" and not the *Kleene* operator (as known from regular expressions (cf. [Aho et al. (1986)])).

It must be also mentioned that the field pattern rules (F) can be used in specifying the array element fields, since the identifier I can contain numbers that are attached to the array name. This meets the way arrays are represented in the object graph model in Section 5.2.1.

Section 4.2 presented various examples of PEP's. However, those examples did not show generally the possible path patterns that can be expressed using the PEP syntax and that are needed to define the denotational semantics of PEP.

[1] A consequence of this assumption is to eliminate the need to define suitable functions that match the given identifier and type patterns.

Table 5.1 lists all possible general syntactic phrases for which a formal semantics is needed.

Table 5.1: Possible syntactic patterns of PEP

$T_1\ id_1\ \ -/\rightarrow\ \ T_2\ id_2$	any path of any length between an object of type T_1 and an object of type T_2
$T_1\ id_1\ \ -*\rightarrow\ \ T_2\ id_2$	any path of one edge only between an object of type T_1 and an object of type T_2 via any association
$T_1\ id_1\ \ -fname\rightarrow\ \ T_2\ id_2$	a path of one edge only between an object of type T_1 and an object of type T_2 via the association that is called $fname$
$R^{n-1}\ T_n\ id_n\ \ -F\rightarrow\ \ T_{n+1}\ id_{n+1}$	any path of any length between an object specified by the left-most relation of a composition of n relations of the above three forms and an object of type T_{n+1}

Note that some of the examples in Section 4.2 used no types in the object pattern part of the path expression patterns. Nevertheless, those situations can be expressed by the patterns in Table 5.1, where the types of objects are induced from the corresponding types in the pointcut header.

5.4 The Semantic World

In order to define the denotational semantics of the path pointcut, it is necessary to specify the following components:

1. **The semantic domains** that are to be used in mapping the syntactic constructs to their semantic meanings.

2. **The auxiliary functions** that are to be defined whenever it is required by the semantics.

3. **The semantic functions** that are to be used in the mapping.

4. **The semantic equations** that show how the functions act on each pattern in the syntactic definition of the language phrases.

In what follows, these four components are described in detail.

5.4.1 The Semantic Domains

The resulting paths from the matching process are represented by *PathRep*, which is defined as a *totally ordered set* of edges that is a subset of E, (the set of edges in the current object graph G). The edges in *PathRep* are ordered based on their occurrence in the path. This occurrence property represents the *total relation* on *PathRep*:

$$\begin{aligned} PathRep \;=\; & \{e_1,\ e_2,\ \ldots,\ e_n\} \text{ where } PathRep \subseteq E \\ & \wedge \forall\ e_j \in PathRep\ \exists\ e_i \in PathRep \text{ s.t.} \\ & \quad target(e_i) = source(e_j) \quad \text{for} \quad 1 \leq i < j \leq n \end{aligned} \tag{5.9}$$

The subscripts $1,\ 2, \ldots,\ n$ represent the order of the edges' occurrences in the path. The first condition is clear, and it ensures that all edges in any valid path representation must be edges in the object graph, i.e. they must be elements of the set E. The second condition in the *PathRep* definition is used to ensure the connectivity property of valid paths in a proper edge' occurrence order. It means: For each edge e_j that occurs later in a valid *PathRep* set, there exists a predecessor edge e_i such that $target(e_i) = source(e_j)$ given that $1 \leq i < j \leq n$.

For example, from Figure 5.1, consider the set:

$p = \{(p1,\ items2,\ c2),\ (c2,\ president,\ e),\ (e,\ familyMembers1,\ s1),$
$(s1,\ familyMembers1,\ e),\ (s1,\ residence,\ a1)\}$

The predecessor edge of $e_4 = (s1,\ familyMembers1,\ e)$ is the edge $e_3 = (e,\ familyMembers1,\ s1)$. Moreover, e_3 is also a predecessor edge of $e_5 = (s1,\ residence,\ a1)$. Similarly, e_2 is the predecessor of e_3 and e_1 is the predecessor of e_2. Hence, p is an ordered-connected valid path representation.

As an example of non-valid paths, consider the set:

$r = \{(p1,\ items2,\ c2),\ (e,\ residence,\ a1),\ (c2,\ president,\ e)\}$

The source node of edge (e, $residence$, $a1$) is e, which is not the target of the only predecessor edge ($p1$, $items2$, $c2$) in r, despite the fact that e is the target of the successor edge ($c2$, $president$, e). I.e., even though the edges in r look like connected, r is not an ordered set of edges, hence, it is not a valid path.

This representation maintains the whole object information in the path including, e.g. the field information and the cycles. Due to the use of sets in representing the paths, an edge cannot occur twice in any path. For example, consider the following path:

$p1 - items1 \rightarrow c2 - president \rightarrow e - familyMembers1 \rightarrow s1$
$-familyMembers1 \rightarrow e - familyMembers1 \rightarrow s1$

This path contains a cycle that contains the nodes e and $s1$. Its $PathRep$ is:

$\{(p1,\ items2,\ c2),\ (c2,\ president,\ e),\ (e,\ familyMembers1,\ s1),\ (s1,\ familyMembers1,\ e)\}$

Even this set represents any such path of infinite length that is caused by the cycle between the `Customer` and the `Employee` objects.

From the definition of $PathRep$, the first semantic domain is derived. It is the set of all paths matching a given path pattern:

$$\mathscr{H} \quad = \quad Set(PathRep) \tag{5.10}$$

The set of all valid $PathRep$'s in the object graph of Figure 5.1 is:

$\mathscr{H} = \{\{(p1,\ items1,\ c1)\},\ \{(c1,\ headquarter,\ a2)\},\ \{(p1,\ items2,\ c2)\},\ \ldots,$

$\{(p1,\ items1,\ c1),\ (c1,\ headquarter,\ a2)\},\ \{(c2,\ divisions1,\ d),\ (d,\ customers1,\ s1)\},\ \ldots,$

$\{(p1,\ items2,\ c2),\ (c2,\ divisions1,\ d),\ (d,\ customers1,\ s1)\}, \{(c2,\ president,\ e),\ (e,\ familyMembers1,\ s1),\ (s1,\ familyMembers1,\ e)\},\ \ldots,$

$\{(c2,\ divisions1,\ d),\ (d,\ customers1,\ s1),\ (s1,\ familyMembers1,\ e),\ (e,\ familyMembers1,\ s1)\},\ \{(p1,\ items2,\ c2),\ (c2,\ president,\ e),\ (e,\ familyMembers1,\ s1),\ (s1,\ familyMembers1,\ e)\},\ \ldots,$

$\{(p1,\ items2,\ c2),\ (c2,\ president,\ e),\ (e,\ familyMembers1,\ s1),\ (s1,\ familyMembers1,\ e),\ (s1,\ residence,\ a1)\},\ \{(c2,\ divisions1,\ d),\ (d,\ customers1,\ s1),\ (s1,\ familyMembers1,\ e),\ (e,\ familyMembers1,\ s1),$

$(e,\ residence,\ a1)\},\ \ldots,$

$\{(p1,\ items2,\ c2),\ (c2,\ divisions1,\ d),\ (d,\ customers1,\ s1),$
$(s1,\ familyMembers1,\ e),\ (e,\ familyMembers1,\ s1),\ (e,\ residence,\ a1)\},$
$\{(p1,\ items2,\ c2),\ (c2,\ divisions1,\ d),\ (d,\ customers1,\ s1),$
$(s1,\ familyMembers1,\ e),\ (e,\ familyMembers1,\ s1),\ (s1,\ residence,\ a1)\}$
$\}$

The first "..." indicates the rest of single-edge paths, the second one indicates the rest of two-edge paths, and so on.

From this set, one can observe that the longest valid path is of length 6. A feasible benefit from this representation is that the set of all possible paths is finite. This ensures the implementation of the path pointcuts in the presence of cycles.

The path representation permits to specify the `PEGraph` objects as sets: if the set of matching paths at a given join point is $h \subseteq \mathscr{H}$ and $h = \{p_1,\ p_2,\ \ldots\ p_n\}$, the corresponding `PEGraph` object is defined as follows:

$$pg \quad = \quad \bigcup_{i=1}^{n} p_i \text{ s.t. } p_i \in h \tag{5.11}$$

The variable pg ranges over the set of all resulting `PEGraph` objects, which are denoted as $\mathscr{PG}$ to represent another semantic domain:

$$\mathscr{PG} \quad = \quad \{pg \mid pg \subseteq E\} \tag{5.12}$$

In the semantic world, G still represents the object graph, since it is the context needed to apply the semantic equation as shown later in this section. Similarly, it is required to have semantic representations for the other components of G. Accordingly, this section maintains the same notations for the set of edges E, the set of nodes V, and the set of types T, which are going to be used as corresponding semantic domains. Hence, the functions that are defined in the object graph model can be used directly in the semantic specification.

Next, it is necessary to map the set of object identifiers O from the abstract syntax of PEP to a corresponding semantic domain. This semantic domain is denoted as O also. Similarly, the `PEGraph` identifiers are elements of the domain GID.

In what follows, the small letters are used to indicate the elements of the domains: $x \in V$, $e \in E$, $t \in T$, $l \in L$, $id \in O$, $gid \in GID$ and $pg \in PG$.

As mentioned earlier, the path pointcut has its own semantics of the parameter bindings and the context exposure mechanisms. Therefore, the denotational semantics must contain a specification of these features. The semantic domain of bindings $\mathscr{B}$ is defined as a set, whose elements are $(id,\ x)$ pairs. Here, id is the identifier that occurs in the path expression pattern and x is the node that represents the object bound to id. In addition to that, the resulting binding set contains a binding of the `PEGraph` object to the corresponding gid. The binding domain is defined as follows:

$$\begin{aligned}\mathscr{B} \;&=\; \{(id_1,\ x_1),\ \ldots,\ (id_n,\ x_n)\} \cup \{(gid,\ pg)\},\\ &\qquad \text{where } \nexists id \in O,\ x_1,\ x_2 \in V \text{ s.t. } (id,\ x_1),\ (id,\ x_2) \in \mathscr{B} \end{aligned} \tag{5.13}$$

The condition in $\mathscr{B}$ ensures that there is no identifier that can be bound to two different objects.

The evaluation of PEP works as follows: The set of matching paths, say h, and the set of bindings, say b, are evaluated first. Then pg is constructed from h such that a single pg corresponds to a single distinct valid set of parameter binding b. The binding of pg is added to b, which is term for exposure to the aspect.

5.4.2 Semantic Auxiliary Functions

This section defines a number of auxiliary functions that operate on the nodes, the paths and the path expression graphs.

The *length* function returns the number of edges in a given path.

$$\begin{aligned} length \;&:\; \mathscr{H} \longrightarrow Number\\ length(p) \;&:\; |p| \;=\; |\{e|\, e \in p\}| \end{aligned} \tag{5.14}$$

To examine the reachability between two given nodes inside a path, one can use the function *reachable*. It takes three arguments, the path p and two nodes x and y. It returns *true*, if there is a subset s of p where x is the source node of the first edge in s and y is the target node of the last edge in s, which means that y is reachable from x in p. Otherwise, the function returns $false$.

$$\begin{aligned} \mathit{reachable} &: \mathscr{H} \times V \times V \longrightarrow \mathit{Boolean} \\ \mathit{reachable}(p,\ x,\ y) &: \mathit{true}, \quad \text{if } \exists s = \{e_1,\ \ldots,\ e_n\} \subseteq p \text{ s.t.} \\ & \quad x = \mathit{source}(e_1)\ \wedge y = \mathit{target}(e_n) \quad \text{for } n \geq 1; \\ &: \mathit{false}, \text{otherwise} \end{aligned} \tag{5.15}$$

In order to get access to the *strongly-connected components* of a given node in a given path, the function *ssc* is used. The *scc* of the node x in the path p is the set of all nodes in p that are reachable from x and vice-versa.

$$\begin{aligned} \mathit{scc} &: \mathscr{H} \times V \longrightarrow \mathit{Set}(V) \\ \mathit{scc}(p,\ x) &: \{x\} \cup \{y \mid \mathit{reachable}(p,\ x,\ y) \wedge \mathit{reachable}(p,\ y,\ x)\} \end{aligned} \tag{5.16}$$

The function *start* obtains the first node that occurs in a path. Each path has only one start node.

$$\begin{aligned} \mathit{start} &: \mathscr{H} \longrightarrow V \\ \mathit{start}(p) &: x = \mathit{source}(e_1),\ \text{where } p = \{e_1,\ e_2,\ \ldots,\ e_n\} \end{aligned} \tag{5.17}$$

Since paths may contain cycles, there might be more than one valid end node of a given cyclic path. For this purpose, the function *end* is provided to obtain the set of such potential end nodes of the path. This set is determined by using the function *scc* that is applied to the target node of the last edge in the path p.

$$\begin{aligned} \mathit{end} &: \mathscr{H} \longrightarrow \mathit{Set}(V) \\ \mathit{end}(p) &: \{x \mid x \in \mathit{scc}(p,\ \mathit{target}(e_n))\},\ \text{where } p = \{e_1,\ e_2,\ \ldots,\ e_n\} \end{aligned} \tag{5.18}$$

The set of matching paths that are constructing a given path expression graph *pg* can be obtained by using the function *paths* that is defined as follows:

$$\begin{aligned} paths &: \mathscr{PG} \longrightarrow Set(\mathscr{H}) \\ paths(pg) &: h, \text{ the set of matching paths constructing } pg \end{aligned} \tag{5.19}$$

The function *combine* concatenates two paths, just by concatenating the first with the second. The result is also a *PathRep*, which is a totally ordered set of edges.

$$\begin{aligned} combine &: \mathscr{H} \times \mathscr{H} \longrightarrow \mathscr{H} \\ combine(\{e_{1_1},\ \ldots e_{1_n}\},\ \{e_{2_1}, \ldots e_{2_m}\}) &: \{e_{1_1},\ \ldots e_{1_n},\ e_{2_1},\ \ldots e_{2_m}\} \end{aligned} \tag{5.20}$$

5.4.3 The Main Semantic Functions

As mentioned earlier, the path pointcut matches the path expression pattern P against the current object graph G in order to find the matching paths. Then, it constructs the `PEGraph` object *pg* that represents the matching paths and consequently, binds the variables in P to the corresponding objects in G. Finally, it exposes *pg* along with the bound objects to the aspect that are given by the set BV.

This functionality is achieved by two main semantic functions: The *matching* function that is denoted by $\mathcal{M}$ and the *selection* function that is denoted by $\mathcal{S}$:

$$\mathcal{M} : PEP \times G \longrightarrow Set(\mathscr{B}) \tag{5.21}$$

$$\mathcal{S} : PEP \times BV \times G \longrightarrow Set(\mathscr{B}) \tag{5.22}$$

The procedure starts by calling function $\mathcal{S}$ with two parameters, the first is the path pointcut and the second is the set of object variables that need to be bound by the pointcut. Then, it calls the matching function $\mathcal{M}$ with the PEP as an argument. The matching function matches the path pattern to G and returns the set of bindings $b' \subseteq \mathscr{B}$ for the `PEGraph` identifier and corresponding bindings for each object identifier of the path pattern. Finally, function $\mathcal{S}$ filters b by removing unneeded bindings according to BV, accordingly combines the `PEGraph` objects and returns the final resulting binding set $b \subseteq \mathscr{B}$. Note that the result is a set of $\mathscr{B}$, which indicates the possibility of multiple valid bindings resulting from pointcut.

It must be mentioned that the PEP matching process may either succeed or fail. A successful matching results in a set of bindings whereas in case of no matching, the PEP returns nothing. The above definition of the set of bindings $\mathscr{B}$ in Equation (5.13) does not consider the failure case. However, this is postponed to a later section in this chapter, when the semantics of a complete pointcut language is presented.

5.4.4 The Semantic Equations

The last part is to specify the semantic equations. For each phrase that may occur in the program (from the abstract syntax of the path expression pointcut, cf. Table 5.1), there is a corresponding semantic equation that explains the phrase meaning by means of semantic functions.

Table 5.2 shows the semantic equations for the path expression pointcut. The notation $\mathcal{M}[\![pep]\!]_G$ is used to denote the application of the matching function $\mathcal{M}$. Similarly, the application of the selection function in the equations is denoted as $\mathcal{S}[\![pep,\ bv]\!]_G$. The equations illustrate the result of applying $\mathcal{S}$ to the path pointcut pep and the set bv as arguments on the context G.

There are four main phrases from the abstract syntax of the path expression pattern that could occur in the code; hence, four variations for $\mathcal{M}$ are considered:

$$\mathcal{M}[\![gid,\ t_1\ id_1\ \ - / \rightarrow t_2\ id_2]\!]_G \tag{5.23}$$

$$\mathcal{M}[\![gid,\ t_1\ id_1\ \ - * \rightarrow t_2\ id_2]\!]_G \tag{5.24}$$

$$\mathcal{M}[\![gid,\ t_1\ id_1\ \ - fname \rightarrow t_2\ id_2]\!]_G \tag{5.25}$$

$$\mathcal{M}[\![gid,\ R^{n-1}\ t_{n-1}\ id_{n-1}\ \ - F \rightarrow t_n\ id_n]\!]_G \tag{5.26}$$

Other forms of the function $\mathcal{M}$ are those where no type information is provided for one or for both variables in the pattern, e.g., $\mathcal{M}[\![pg,\ id_1\ - fname\ \rightarrow\ id_2]\!]_G$.

Table 5.2: Denotational semantics of PEP

// Selection

$\mathcal{S} \quad : \quad PEP \times BV \times G \longrightarrow Set(\mathscr{B})$

$\mathcal{S}[\![path(gid,\ pep),\ bv]\!]_G \ = \ \{b \mid b = \{(gid,\ pg)\} \cup b'\}$ where

$\quad pg = \bigcup_{i=1}^{m}\ pg_i \ \wedge \ b' = \bigcup_{i=1}^{m}\ b_i - \{(id,\ x),\ (gid_1,\ pg_i)\}$

$\qquad$ for $m \leq |\mathcal{M}[\![gid,\ pep]\!]_G| \ \wedge \ id \ni bv \ \wedge \ b_i \in \mathcal{M}[\![gid,\ pep]\!]_G$

// Matching

$\mathcal{M} \quad : \quad PEP \times G \longrightarrow Set(\mathscr{B})$

$\mathcal{M}[\![gid,\ t_1\ id_1\ \ - / \rightarrow t_2\ id_2]\!]_G$

$= \{b \mid b = \{(gid,\ pg),\ (id_1,\ x_1),\ (id_2,\ x_2)\}$

$\quad \wedge \forall p \in paths(pg) \Rightarrow$

$\qquad (x_1 = start(p) \wedge \ x_2 \in end(p) \wedge \ length(p) \geq 1$

$\qquad\quad \wedge \ t_1 \in types(atype(x_1)) \wedge \ t_2 \in types(atype(x_2)))\}$

$\mathcal{M}[\![gid,\ t_1\ id_1\ \ - * \rightarrow t_2\ id_2]\!]_G$

$= \{b \mid b \in \mathcal{M}[\![gid,\ t_1\ id_1\ \ - / \rightarrow t_2\ id_2]\!]_G$

$\quad \wedge(\ (gid,\ pg) \in b \ \wedge \ \forall p \in paths(pg) \Rightarrow length(p) = 1)\}$

$\mathcal{M}[\![gid,\ t_1\ id_1\ \ - fname \rightarrow t_2\ id_2]\!]_G$

$= \{b \mid b \in \mathcal{M}[\![gid,\ t_1\ id_1\ \ - * \rightarrow t_2\ id_2]\!]_G$

$\quad \wedge(\ (gid,\ pg) \in b \ \wedge \ \forall p \in paths(pg) \ \wedge \ \forall e \in p \Rightarrow label(e) = fname)\}$

$\mathcal{M}[\![gid,\ R^{n-1}\ t_{n-1}\ id_{n-1}\ \ - F \rightarrow t_n\ id_n]\!]_G$

$= \{b \mid b = b_1 \cup b_2$

$\quad \wedge b_1 = \mathcal{M}[\![gid_1,\ R^{n-1}\ t_{n-1}\ id_{n-1}]\!]_G \wedge b_1 = \mathcal{M}[\![gid_2,\ t_{n-1}\ id_{n-1}\ - F \rightarrow t_n\ id_n]\!]_G$

$\quad \wedge$ for $(gid_1,\ pg_1) \in b_1 \wedge (gid_2,\ pg_2) \in b_2$

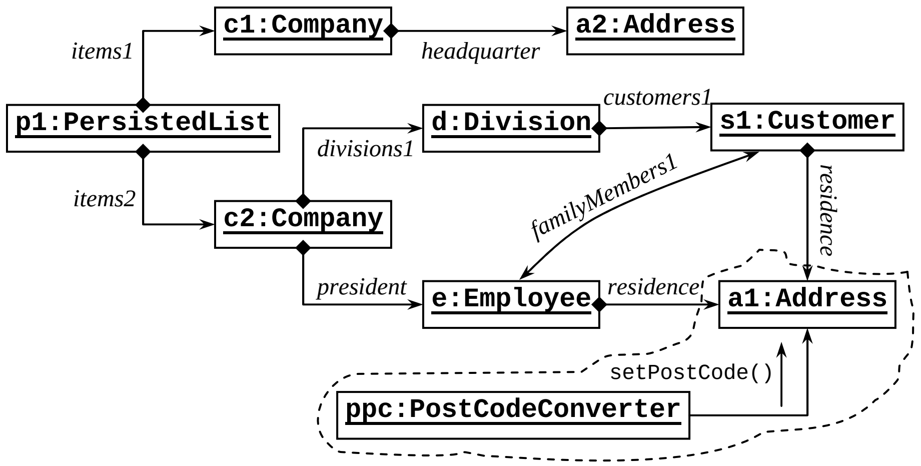
$\qquad \Rightarrow (gid,\ pg = pg_1 \cup pg_2)$ where $\forall p_1 \in paths(pg_1) \wedge p_2 \in paths(pg_2)$

$\qquad\qquad \Rightarrow combine(p_1,\ p_2) \in paths(pg)$ if $start(p_2) \in end(p_1)\}$

The evaluation would be the same since the types of these object identifiers must be mentioned in the pointcut header. However, one can consider that these variables are of the general supertype `Object`. I.e., this pattern is the same as: $Object\,id_1 - fname \rightarrow Objectid_2$, which is equal to 5.25.

5.5 Example of the Semantics Usage

In this section, a concrete example of the usage of the formal semantics presented above is given. Recall the motivating example in Section 4.3.1. Figure 5.2 illustrates the same object graph where the `postcode` field of the `Address` object `a1` is being changed. This join point is selected by the pointcut `pc` of the figure.



```
pointcut pc(PEGraph gid, Object o):
  set(* *.*) && target(o) && path(gid, PersistedList p -/-> o);
```

Figure 5.2: A field set join point on object `a1` selected by pointcut `pc`

The object graph of Figure 5.2 consists of the components as shown in the example of Section 5.2.2.

Equation (5.23) is used next according to the path expression pattern in the above pointcut. First of all, the matching paths are to be specified. There are eight matching paths from the set of all valid paths $\mathscr{H}$ in G (cf. $\mathscr{H}$ in page 122):

p_1={($p1$, $items2$, $c2$), ($c2$, $president$, e), (e, $residence$, $a1$)}
p_2={($p1$, $items2$, $c2$), ($c2$, $divisions1$, d), (d, $customers1$, $s1$),
($s1$, $residence$, $a1$)}
p_3={($p1$, $items2$, $c2$), ($c2$, $divisions1$, d), (d, $customers1$, $s1$),
($s1$, $familyMembers1$, e), (e, $residence$, $a1$)}
p_4={($p1$, $items2$, $c2$), ($c2$, $divisions1$, d), (d, $customers1$, $s1$),
($s1$, $familyMembers1$, e), (e, $familyMembers1$, $s1$), ($s1$, $residence$, $a1$)}
p_5={($p1$, $items2$, $c2$), ($c2$, $divisions1$, d), (d, $customers1$, $s1$),
($s1$, $familyMembers1$, e), (e, $familyMembers1$, $s1$), (e, $residence$, $a1$)}
p_6={($p1$, $items2$, $c2$), ($c2$, $president$, e), (e, $familyMembers1$, $s1$),
($s1$, $residence$, $a1$)}
p_7={($p1$, $items2$, $c2$), ($c2$, $president$, e), (e, $familyMembers1$, $s1$),
($s1$, $familyMembers1$, e), (e, $residence$, $a1$)}
p_8={($p1$, $items2$, $c2$), ($c2$, $president$, e), (e, $familyMembers1$, $s1$),
($s1$, $familyMembers1$, e), ($s1$, $residence$, $a1$)}

These matching paths construct the set h:

$$h = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$$

The `PEGraph` object `pg` is constructed from the union of all paths in h:

$$\begin{aligned} pg = \{&(p1, items2, c2), (c2, president, e), (e, residence, a1), (c2, \\ &divisions1, d), (d, customers1, s1), (s1, familyMembers1, e), \\ &(e, familyMembers1, s1), (s1, residence, a1)\}\end{aligned}$$

Then, for each path in h, the relevant auxiliary functions *start*, *end*, and *length* are performed. For example, the executions of these functions on p_1 and p_4 are as follows:

$$\begin{aligned} start(p_1) &= p1 \\ end(p_1) &= \{a1\} \\ length(p_1) &= 3 \end{aligned}$$

$$\begin{aligned} start(p_4) &= p1 \\ end(p_4) &= \{a1\} \\ length(p_4) &= 6 \end{aligned}$$

This shows that each path in pg between a `PersistedList` object and the dirty `Address` object is starting from object $p1$ and ending at object $a1$. At the same time, each path is of length greater than one. Moreover, the conditions on types of $p1$ and $a1$ are fulfilled (cf. Section 5.2.2). Here, $t_1 = PersistedList$ and

$t_2 = Object$, which are part of the pointcut header.

$$\begin{aligned} atypes(p1) &= PersistedList \\ types(PersistedList) &= \{PersistedList, Object\} \end{aligned}$$

$$\begin{aligned} atype(a1) &= Address \\ types(Address) &= \{Address, Object\} \end{aligned}$$

Then, all required conditions are checked and fulfilled. Hence, the application of Equation (5.23) gives the following result:

$$\mathcal{M}[\![gid,\ PersistedList\ p - / \rightarrow Object\ o]\!]_G$$
$$= \{\ \{(gid,\ pg),\ (p,\ p1),\ (o,\ a1)\}\}$$

There is only one resulting valid binding set, from which function $\mathcal{S}$ will remove the binding of the variable p, since the poincut `pc` requires bindings for the variables `gid` and `o`. Moreover, there is no need to combine any path expression graphs, since $\mathcal{M}$ results only in a single set of bindings:

$$\mathcal{S}[\![\texttt{path}(gid,\ PersistedList\ p - / \rightarrow Object\ o),\ \{gid,\ o\}]\!]_G$$
$$= \{\ \{(gid,\ pg),\ (o,\ a1)\}\}$$

Figure 5.3 shows the matching `PEGraph` object `pg`, which is included in the exposed bindings set.

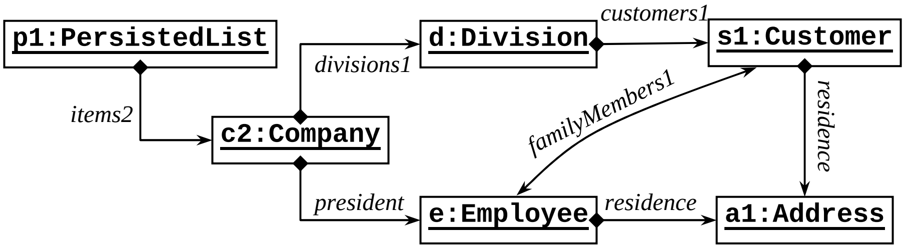


Figure 5.3: The resulting `PEGraph` from pointcut `pc` above

Due to the mathematical description of the components in terms of set theory and the usage of variables, the description of the meaning of the given PEP is no more ambiguous. It describes precisely the paths, the `PEGraph` objects, and the bindings. It clarifies the relationships between these components. Moreover, it also describes how actual types are resolved. As a consequence, the implementation of the concept becomes much easier than relying on informal descriptions.

5.6 PEP Semantics in a Complete Pointcut Language

For the purpose of completing the semantics of PEP, it is necessary to integrate the above derived semantics into the current pointcut languages. Such integration will remove any ambiguity from the result of the pointcut that uses the PEP designator.

The above presented semantics of PEP shows that the result of PEP at a given join point is a set of distinct valid bindings. This exactly meets the semantics of the result of any pointcut designator in aspect-oriented programming [Wand et al. (2004)]. The only difference is that using PEP in a pointcut may result in multiple valid distinct bindings instead of one set of bindings.

Until this point, the thesis considered the successful matching case of PEP with a non-empty set of bindings. In this section, the remaining cases are considered, i.e. successful matching with empty bindings, e.g. with non-binder pointcuts such as the `execution` pointcut designator in AspectJ as well as the failure case.

In order to perform the integration, a base language is to be chosen. This thesis will consider a minimal language of AspectJ called Aspect Sound Box (ASB) as the base language. A denotational formal semantics for ASB is proposed by Wand et al. (2004). A part of that semantics will be extended in this section.

For the purpose of understandability, this thesis will use the AspectJ syntax instead of the ABS syntax given in [Wand et al. (2004)]. For example, the Figure 5.4 shows a valid advice declaration in ASB and how it is rendered to AspectJ syntax.

For the purpose of self-containedness, Section 5.6.1 gives a short introduction to the semantics of the base language ASB. Then, Section 5.6.2 extends this semantics by adding PEP to its syntax and defines the semantics of the resulting pointcut language.

5.6.1 The Semantics of the Base Language

This section introduces a part of the formal semantics given in [Wand et al. (2004)] that is enough to investigate the integration of the binder PEP pointcut.

```
// ASB Syntax ...
(around
  (and
    (pcalls int fact (int))
    (args (int x)))
  // ... before the join point
  (proceed x)
  // ... after the join point
)

// AspectJ Syntax
void around(int x): call(* int *.fact(..)) && args(x) {
  // ... before the join point
  proceed(x);
  // ... after the join point
}
```

Figure 5.4: ASB vs. AspectJ syntax

This part is restricted to the call and execution pointcut designators. It does not consider the join points that are resulting from running the advice, neither it considers the control stack. Moreover, the method names are assumed to be exact names. Consequently, the manipulation of name patterns matching is ignored.

5.6.1.1 Join Points

A join point refers to an event during the execution of the program, and it has distinguishing properties as stated earlier. These properties include the kind, the method name, the target object, and the return type. The join point domain JP is defined as follows:

$$
\begin{array}{lll}
jp & \in & JP \\
JP & ::= & \langle k,\ mname,\ targetobj,\ rtype \rangle \\
k & ::= & mcall \mid mexec
\end{array}
$$

In fact, the join points may have other properties such as the `this` object and arguments. These properties are eliminated for the purpose of simplicity, since one can consider the target object property as an analogous to them. For example, the join point $\langle mcall, m, t, T\rangle$ represents a call to the method m on object t and whose return type is T.

5.6.1.2 Pointcuts

The selection of the join points is achieved by using a combination of pointcut designators in the declaration of the advice. Each pointcut specifies the set of join points to which the advice is applicable. The matching process of a pointcut with a join point either succeeds with a set of bindings or fails. The following represents the selected pointcut grammar of ABS:

$$\begin{array}{lcl} \phi,\ \psi & ::= & \texttt{execution}(R\ T.mname) \\ & | & \texttt{call}(R\ T.mname) \\ & | & \texttt{target}(id) \\ & | & \phi\ \texttt{\&\&}\ \psi \\ & | & \phi\ \texttt{||}\ \psi \\ & | & \texttt{!}\ \phi \end{array}$$

5.6.1.3 Variable Bindings

The result of matching a pointcut is taken from the following binding domain $\mathscr{B}$:

$\mathscr{B}$	$::=$	$\{\}$	succeeds with empty set of bindings
	$\vert$	$\{(id,\ x)\}$	succeeds with the binding for the target object x to the variable id
	$\vert$	$Fail$	fails

If b, b_1, $b_2 \subseteq \mathscr{B}$, the result from composed pointcuts is given according to the following binding logic rules:

Binding Logic

$$\begin{array}{l} \neg Fail = \{\} \quad \neg b = Fail \\ b \vee Fail \ = \ Fail \vee b \ = \ b \quad b_1 \vee b_2 \ = \ b_1 \\ b_1 \wedge b_2 \ = \ b_2 \wedge b_1 \ = \ b_1 \cup b_2 \\ b \wedge Fail \ = \ Fail \wedge b \ = \ Fail \end{array}$$

This logic shows the operations of $\wedge$, $\vee$, and $\neg$ on the bindings and pointcut results. Note that both $\wedge$ and $\vee$ are short-cuttings, so that $\vee$ prefers its first argument, i.e. the result of $b_1 \vee b_2$ in this semantics is determined by b_1 if $b_1 \neq Fail$.

Moreover, it is necessary to mention that the above pointcut language has only the `target` pointcut as a binder, which in turn results in a binding set that has only one set of bindings to the target object. So, the union $\cup$ will result in any case to such a single-element binding, and there is no need to redefine how bindings are combined in ASB. However, the combine functionality in the presence of PEP differs from this one, and it will be defined later on.

5.6.1.4 Semantic Equations

The pointcut `execution`$(R\ T.mname)$ matches the join point $jp = \langle k,\ mname',\ targetobj,\ rtype\rangle$ if $k = mexec$, $mname = mname'$, $targetobj$ is of type T, and R equals or is a supertype of $rtype$. Then, the result is an empty set of bindings $\{\}$. A similar case is applied for the `call`$(R\ T.mname)$ pointcut except that $k = mcall$, hence, call pointcut will not be considered furthermore.

Similarly, the pointcut `target`(id) matches the jp if $targetobj$ is of type T and if so, the result is the binding set $\{(id,\ targetobj)\}$.

Matching a join point $jp = \langle k,\ mname',\ targetobj,\ rtype\rangle$ by a pointcut in ASB is determined by the matching function $\mathcal{M}$. The semantic equations that show how this function operates are as follows:

SEMANTIC EQUATION

$$//Matching\mathcal{M}\ :\ PC\ \times\ JP\ G\ \longrightarrow\ Set(\mathscr{B})$$

$$\mathcal{M}[\![\texttt{execution}(R\ T.mname),\ \langle k,\ mname',\ targetobj,\ rtype\rangle]\!]$$
$$= \begin{cases} \{\} & \text{if } k = mexec\ \wedge\ mname = mname' \\ & \quad \wedge\ targetobj \text{ is of type }\ \wedge\ R \text{ is a supertype of } rtype \\ Fail & \text{otherwise} \end{cases}$$

$$\mathcal{M}[\![\texttt{target}(id),\ \langle k,\ mname',\ targetobj,\ rtype\rangle]\!]$$
$$= \begin{cases} \{\{(id,\ targetobj)\}\} & \text{if } targetobj \text{ is of type } T \\ Fail & \text{otherwise} \end{cases}$$

$$\begin{aligned}
&\mathcal{M}[\![\phi \;\texttt{\&\&}\; \psi,\ \langle k,\ mname,\ targetobj,\ rtype\rangle]\!] = \mathcal{M}[\![\phi]\!] \wedge \mathcal{M}[\![\psi]\!]\\
&\mathcal{M}[\![\phi \;\texttt{||}\; \psi,\ \langle k,\ mname,\ targetobj,\ rtype\rangle]\!] = \mathcal{M}[\![\phi]\!] \vee \mathcal{M}[\![\psi]\!]\\
&\mathcal{M}[\![\texttt{!}\phi,\ \langle k,\ mname,\ targetobj,\ rtype\rangle]\!] = \neg\mathcal{M}[\![\phi]\!]
\end{aligned}$$

5.6.2 Integrating PEP with Aspect SoundBox (ASB)

The integration of PEP with the above semantics of ASB is achieved by two steps: First, the syntax of ASB pointcut language will be extended by the syntax of PEP. Second, the denotational semantics of the resulting pointcut language will be derived.

The following grammar illustrates the extended syntax. Note that the four syntactic phrases from Table 5.1 are given. Moreover, the last path expression pattern is concrete.

ExtendedPointcutLanguage

$$\begin{array}{lll}
\phi,\ \psi & \in & PC\\
\phi,\ \psi & ::= & \texttt{execution}(R\ T.mname)\\
 & | & \texttt{target}(id)\\
 & | & \texttt{path}(gid,\ T_1\ id_1 - / \rightarrow T_2\ id_2)\\
 & | & \texttt{path}(gid,\ T_1\ id_1 - * \rightarrow T_2\ id_2)\\
 & | & \texttt{path}(gid,\ T_1\ id_1 - f \rightarrow T_2\ id_2)\\
 & | & \texttt{path}(gid,\ T_1\ id_1 - F \rightarrow T_2\ id_2, \ldots, -F \rightarrow T_n\ id_n)\\
 & | & \phi\ \texttt{\&\&}\ \psi\\
 & | & \phi\ \texttt{||}\ \psi\\
 & | & \texttt{!}\ \phi
\end{array}$$

The next step is to obtain a common binding domain for this pointcut language. The binding domain $\mathscr{B}$ that is defined in Section **??** is slightly modified to represent the successful as well as the failure matching result of the pointcuts at a given join point as follows:

BINDINGDOMAIN

$\mathscr{B}$	$::=$	$\{\}$	succeeds with no bindings
	$\mid$	$\{(id,\ x)\}$	succeeds with the binding for the parameter *id*
	$\mid$	$\{(id_1, x_1), \ldots, (id_n, x_n)\} \cup \{(gid, pg)\}$	succeeds with a binding for the parameters of `target` and `path` pointcuts
	$\mid$	$Fail$	fails

Similarly, the semantic functions that deal with the evaluation of the pointcuts need to be modified. The matching function $\mathcal{M}$ matches the pointcut ϕ that is making use of the PEP at the join point jp. If the matching process succeeds, $\mathcal{M}$ returns a set of distinct valid bindings $b \subseteq \mathscr{B}$. This set will be passed to the selection function $\mathcal{S}$, which in turn eliminates the unnecessary bindings and performs any needed combination of the resulting `PEGraph` objects. Since this semantics operates on objects, the object graph G is still considered as the context of these functions.

As noted above, it is necessary to modify the binding logic of the complete pointcut language. This is because PEP normally results in more than one distinct binding set, therefore, the union of the resulting bindings for "`&&`" pointcut has other semantics. If $b \subseteq \mathscr{B}$, $\xi, \xi_1, \xi_2 \subseteq Set(\mathscr{B})$ holds, the following rules define the new semantics of the result binding from pointcut composition.

BINDING LOGIC WITH PEP

$$\neg Fail = \{\} \quad \neg\xi = Fail$$
$$\xi \vee Fail \ = \ Fail \vee \xi \ = \ \xi \quad \xi_1 \vee \xi_2 \ = \ \xi_1$$
$$\xi \wedge Fail \ = \ Fail \wedge \xi \ = \ Fail$$
$$\xi_1 \wedge \xi_2 \ = \ \xi_2 \wedge \xi_1 = \ \{b | b = b_i \cup b_j \ \forall b_i \in \xi_1 \ \wedge \ \forall b_j \in \xi_2\}$$

Note that the resulting $b \in \xi$ in the last rule is a valid binding set since each ξ is a subset of $Set(\mathscr{B})$.

The semantic equations of PEP in Table 5.2 need to be modified to allow the matching of the method-execution join points by adding the condition (k =

mexec). Moreover, these equations have to be modified to express the failure as a possible result of matching. For example, the first semantic equation of Table 5.2 will be expressed as follows:

SEMANTICEQUATIONS

$$\begin{array}{l}
//Matching\mathcal{M} \ : \ PC \ \times \ JP \ \times \ G \ \longrightarrow \ Set(\mathscr{B}) \\
\mathcal{M}[\![\texttt{path}(gid,\ T_1\ id_1 - / \rightarrow T_2\ id_2),\ \langle k, mname, target, rtype\rangle]\!]_G \\
= \{b \mid b = \{(gid, pg), (id_1, x_1), (id_2, x_2)\} \subseteq \mathscr{B}\} \\
\quad\quad \text{if } k = mexec \ \wedge \ \forall p \in paths(pg) \Rightarrow \\
\quad\quad (\ x_1 = start(p) \wedge \ x_2 \in end(p) \wedge \ length(p) \geq 1 \\
\quad\quad\quad \wedge\ T_1 \in types(atype(x_1)) \wedge \ T_2 \in types(atype(x_2))\ ) \\
\quad Fail \text{ otherwise} \\
//Selection\mathcal{S} \ : \ PC \times \ JP \ \times \ BV \times G \longrightarrow Set(\mathscr{B}) \\
\mathcal{S}[\![\texttt{pc}, bv]\!]_G = \{b \mid b = \{(id_1, x_1), \ldots (id_n, x_n)\}\} \text{ where} \\
\quad (id_i \in GID \ \wedge \ x_i \in \mathscr{PG}) \ \vee \ (id_i \in bv \wedge \ x_i \in V)
\end{array}$$

5.6.2.1 Example 1

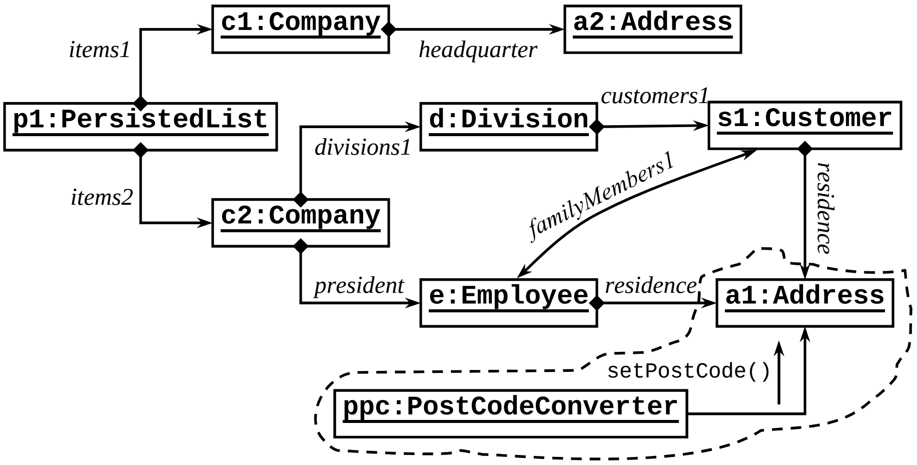
Consider again the example given in Section 5.5. The object graph in Figure 5.5 shows that the message `a1.setPostCode()` is issued by object `pcc`.

The pointcut `perObjChg` in Figure 5.5 captures this join point. As seen in Section 5.5, all matching paths to the path pattern of pointcut `perObjChg` start from the list `p1` and end at the object `a1`. Hence, there is only one distinct parameter binding that corresponds to a single `PEGraph`:

$$\begin{array}{l}
pg = \{(p1, items2, c2), (c2, divisions1, d), (c2, president, e), \\
\quad (d, customers1, s1), (e, familyMembers1, s1), (e, residence, a1), \\
\quad (s1, familyMembers1, e), (s1, residence, a1)\}
\end{array}$$

Then, the result of the pointcut `perObjChg` is determined by the following:

$$\begin{array}{l}
\mathcal{M}[\![\texttt{execution}(*\ *.setPostCode)\ \texttt{\&\&}\ \texttt{target}(o)\ \texttt{\&\&} \\
\quad \texttt{path}(gid,\ PersistedList\ p - / \rightarrow Object\ o), \\
\quad\quad \langle mexec, setPostCode, a1, Object\rangle]\!]_G \\
= \mathcal{M}[\![\texttt{execution}(*\ *.setPostCode), \langle mexec, setPostCode, a1, Object\rangle]\!]_G \\
\wedge\ \mathcal{M}[\![\texttt{target}(o),\ \langle mexec, setPostCode, a1, Object\rangle]\!]_G \\
\wedge\ \mathcal{M}[\![\texttt{path}(gid,\ PersistedList\ p - / \rightarrow Object\ o), \\
\quad \langle mexec, setPostCode, a1, Object\rangle]\!]_G
\end{array}$$



```
pointcut perObjChg(PEGraph gid, Object o):
  target(o) && execution(* *.setPostCode(*)) &&
  path(gid, PersistedList p -/-> o
```

Figure 5.5: Address object `a1` is being updated

$= \{\} \cup \{\{(o, a1)\}\} \cup \{\{(gid, pg), (p, p1), (o, a1)\}\}$
$= \{\{(gid, pg), (p, p1), (o, a1)\}\}$

$\mathcal{S}[\![\mathsf{pc}, \{gid, o\}]\!]_G = \{\ \{(gid,\ pg),\ (o,\ a1)\}\}$

Again, the resulting `PEGraph` is shown in Figure 5.6.

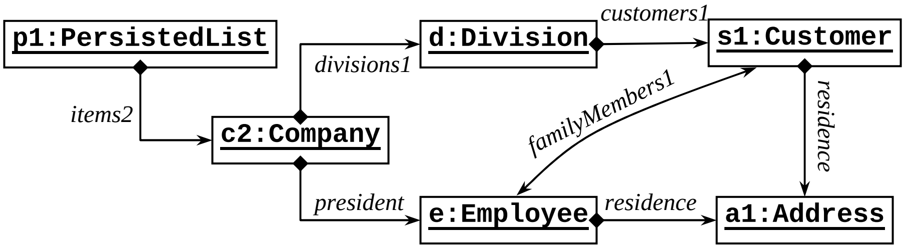
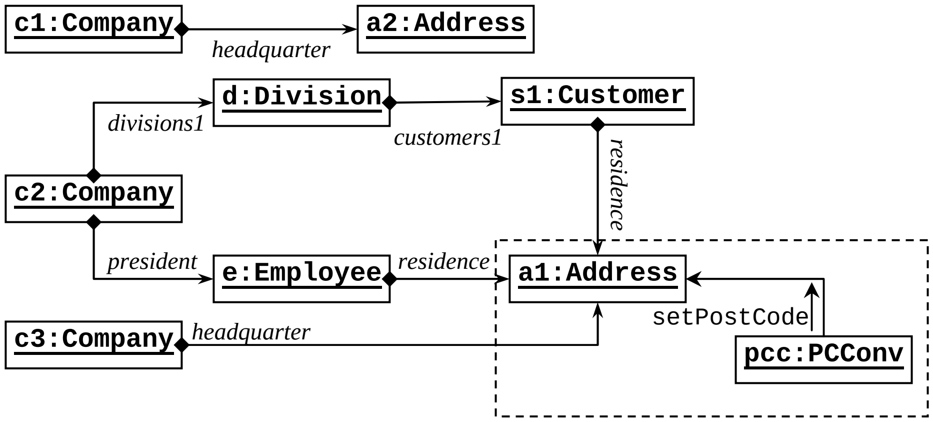


Figure 5.6: Resulting `PEGraph` from pointcut `perObjChg`

5.6.2.2 Example 2

In the previous example, the function $\mathcal{M}$ returns one resulting set of bindings, and there was no need to combine the resulting graph and the bindings. The following example illustrates the combine operation.



```
pointcut comObjChg(PEGraph gid, Object o):
  target(o) && execution(* *.setPostCode(*))
  && path(gid, Company c -/-> o);
```

Figure 5.7: A shared `Address` object between two companies

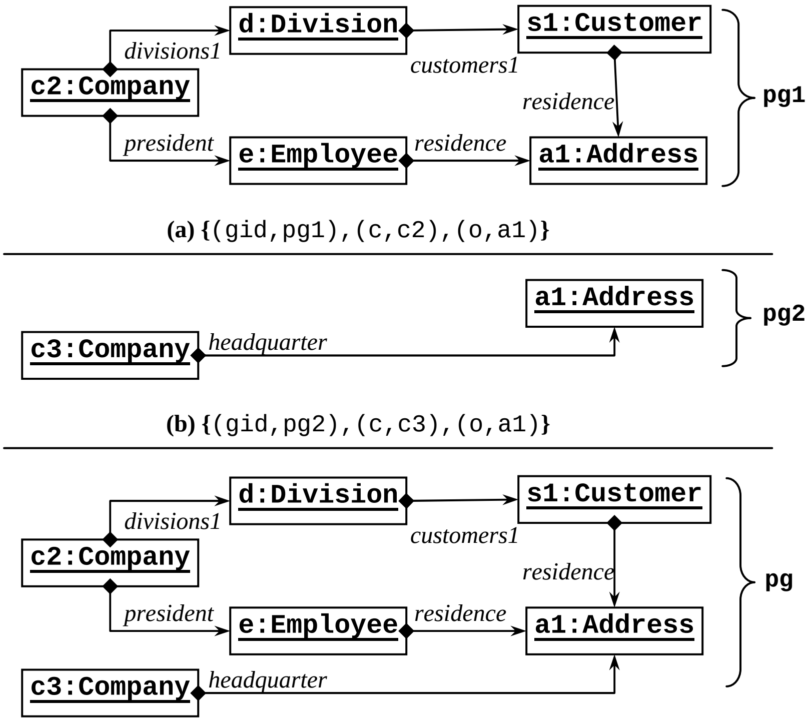
Figure 5.7 depicts a collaboration diagram where two companies, `c2` and `c3`, share the same address object `a1`. Assume again that `a1` is changed by invoking the method `setPostCode`.

Consider the pointcut `comObjChg` in the figure. The `execution` pointcut matches the join point $jp = \langle \mathit{mexec}, \mathit{setPostCode}, a1, \mathit{Object} \rangle$ and results in the empty binding list $\{\}$. The result from the `target` pointcut is the binding of the variable `o` to the target object of jp: $\{\{(o, a1)\}\}$. With respect to PEP, there are three matching paths:

$p_1 = \{(c2, \mathit{divisions}1, d), (d, \mathit{customers}1, s1), (s1, \mathit{residence}, a1)\}$
$p_2 = \{(c2, \mathit{president}, e), (e, \mathit{residence}, a1)\}$
$p_3 = \{(c3, \mathit{headquarter}, a1)\}$

The start node of p_1 and p_2 is the node $c2$ and both also end at node $a1$. Hence, p_1 and p_2 correspond to a single distinct binding $\{(\mathit{gid}, \mathit{pg}1), (c, c2), (o, a1)\}$. This means these two paths represent a single path expression graph $\mathit{pg}1 \in \mathscr{PG}$ that is depicted in Figure 5.8-(a):

$$\begin{aligned} \mathit{pg}1 = \{&(c2, \mathit{divisions}1, d), (d, \mathit{customers}1, s1), (s1, \mathit{residence}, a1), \\ &(c2, \mathit{president}, e), (e, \mathit{residence}, a1)\} \end{aligned}$$



(a) `{(gid,pg1),(c,c2),(o,a1)}`

(b) `{(gid,pg2),(c,c3),(o,a1)}`

(c) `{(gid,pg),(o,a1)}`

Figure 5.8: The result of pointcut `comObjChg`

The third path, p_3, starts at $c3$ and ends at $a1$ and corresponds to another distinct bindings: $\{(gid, pg2), (c, c3), (o, a1)\}$, where $pg2$ equals p_3 as shown in Figure 5.8-(b).

The matching function results in the following:

$\mathcal{M}[\![\texttt{path}(gid,\ Company\ c - / \rightarrow Object\ o),$
$\quad\langle mexec, setPostCode, a1, Object\rangle]\!]_G$
$= \{\{(gid, pg1), (c, c2), (o, a1)\}, \{(gid, pg2), (c, c3), (o, a1)\}\ \}$

The matching function continues the evaluation of the whole pointcut and passes the results to $\mathcal{S}$:

$\mathcal{M}[\![\texttt{execution}(*\ *.setPostCode)\ \texttt{\&\&}\ \texttt{target}(o)\ \texttt{\&\&}$
$\quad\texttt{path}(gid,\ Company\ c - / \rightarrow Object\ o),$
$\qquad\langle mexec, setPostCode, a1, Object\rangle]\!]_G$

$$\begin{aligned}
&= \mathcal{M}\llbracket \texttt{execution}(*\ *.setPostCode), \langle mexec, setPostCode, a1, Object\rangle \rrbracket_G \\
&\wedge\ \mathcal{M}\llbracket \texttt{target}(o),\ \langle mexec, setPostCode, a1, Object\rangle \rrbracket_G \\
&\wedge\ \mathcal{M}\llbracket \texttt{path}(gid,\ Company\ c - / \rightarrow Object\ o), \\
&\quad \langle mexec, setPostCode, a1, Object\rangle \rrbracket_G \\
&= \{\ \{\} \cup \{(o, a1)\} \cup \{(gid, pg1), (c, c2), (o, a1)\}, \\
&\quad \{\} \cup \{(o, a1)\} \cup \{(gid, pg2), (c, c3), (o, a1)\}\ \} \\
&= \{\ \{(gid, pg1), (c, c2), (o, a1)\},\ \{(gid, pg2), (c, c3), (o, a1)\}\ \}
\end{aligned}$$

$$\mathcal{S}\llbracket \texttt{pc}, \{gid, o\} \rrbracket_G = \{\ \{(gid,\ pg),\ (o,\ a1)\}\}$$

These two binding sets must be filtered according to the set of bound variables $bv = \{gid, o\}$. Since `a1` is shared by $pg1$ and $pg2$, function $\mathcal{S}$ combine both into one graph pg (Figure 5.8-(c)) that is to be bound to the variable gid.

5.7 Typing Issues

In this section, a simple type system for the path pointcut is proposed. First, two parametric types for the resulting bindings list of PEP and its resulting `PEGraph` are defined. Then, the static typing rules of PEP are given along with some examples that illustrate the usage of these typing rules.

5.7.1 A Generic Type for Parameter Binding Lists

In order to avoid the unsafe usage of type-casting the bound variables inside the ordering method, it is possible to utilize the type information of the pointcut parameters. Such requirement can be provided with the help of generic types. This is because it is sufficient to have a simple data structure that is responsible only for holding the types in a proper order. Figure 5.9 illustrates the interface of the `PBList`.

The two parameters of the ordering method are of the generic type `PBList`. Each `PBList` object represents a single variable binding set in the order given by the pointcut header. `PBList` interface provides two public methods, `current()` that returns the bound object at the current item of the `PBList`, and `next()` that gives the access to the next element(s) in the `PBList` object.

For example, consider the following pointcut:

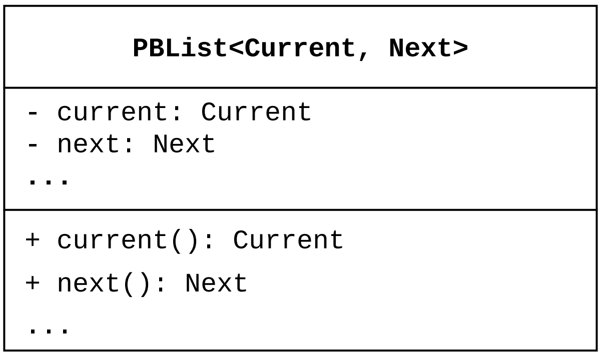


Figure 5.9: `PBList` interface

```
pointcut pc(PEGraph g, Division d, Customer c, Address a):
  path(g, d -*-> c -*-> a) ...;
```

The code fragments in Figure 5.10 define an object of the corresponding `PBList` type to pointcut `pc` and then extracts the bound objects from it.

```
PBList<PEGraph, PBList<Division, PBList<Customer, Address>>> pbList;

// extract the bound objects ...
PEGraph  g = pbList.current();
Division d = pbList.next().current();
Customer c = pbList.next().next().current();
Address  a = pbList.next().next().next().current();
```

Figure 5.10: An example of `PBList`

Note that there is no need to type casting since the `PBList` provides the right types. On the other hand, all object information needed for the ordering is available in the `PBList` objects, which means that it is rarely the case that developers will face a problem to ensure the termination of their ordering methods.

As a complete example of using `PBList`, consider the code in Figure 5.11. The ordering method `addrChg` has two parameters `pbList1` and `pbList2` of type `PBList`. Inside this method, the second objects that are of type `Customer` are retrieved without casting and their `balance` fields are compared to solve the ordering example in Section 4.2.2.5.

```
public boolean
       addrChg(PBList<PEGraph,PBList<Customer,Address>> pbList1,
               PBList<PEGraph,PBList<Customer,Address>> pbList2)
{
  Customer cust1 = pbList1.next().current();
  Customer cust2 = pbList2.next().current();
  return cust1.getBalance() > cust2.getBalance();
}

before(PEGraph g, Customer c, Address a): set(* *) && target(a) &&
  path(g, Division d -/-> c -/-> a) orderBy(this.addrChg) {
  // ...
}
```

Figure 5.11: An example of using PBList type in an ordering method

5.7.2 A Generic Type for Path Expression Graphs

The path expression graph type is defined to include all types that occur in the path pattern. Its interface is given in Figure 5.12. The PEGraph is the simple graph and it is extended by ExPEG.

The PEGraph object is the result of matching path patterns that use only field-name associations or wildcard "*" associations. This reflects the fact that there are only direct relationships (paths of one edge) between the start and the end parts of the graph.

For example, for the pattern (A a -fname-> B b -*-> C c), the resulting path graph g is of type PEGraph<A, PEGraph<B, C>>. The statement g.start() returns a read-only list of objects of type A that represent the start nodes of the graph g. The statement g.end() returns a read-only list that contains only one graph object, say e.g. g1 of type PEGraph<B, C>. The statement g.end().start() returns a read-only list of objects of type B that represent the start nodes of the graph g1. Finally, the statement g1.end() returns a read-only list of objects of type C, which represent the end nodes of the graph g1. Note that a read-only list is used here, so that the programmer cannot mutate it.

On the other hand, the objects of ExPEG have inner objects between the start and the end parts of the graph. This type should be used for the indirect relationships that are specified by the wildcard "/".

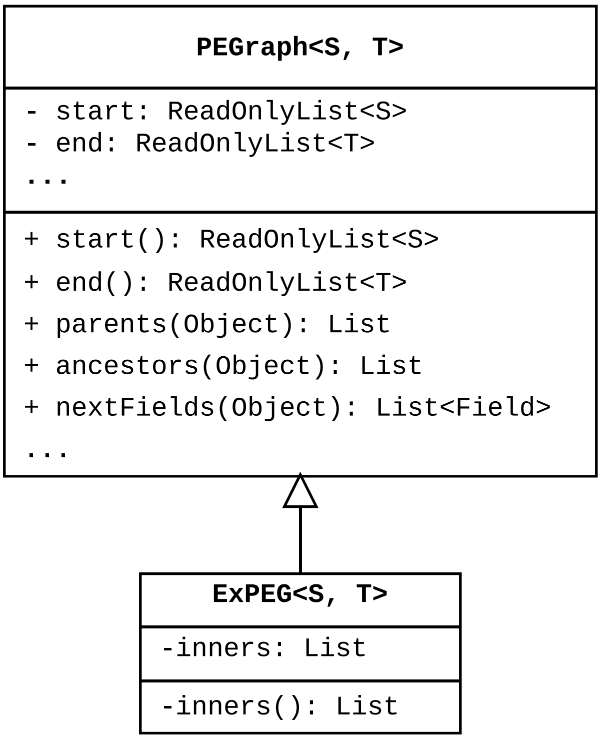


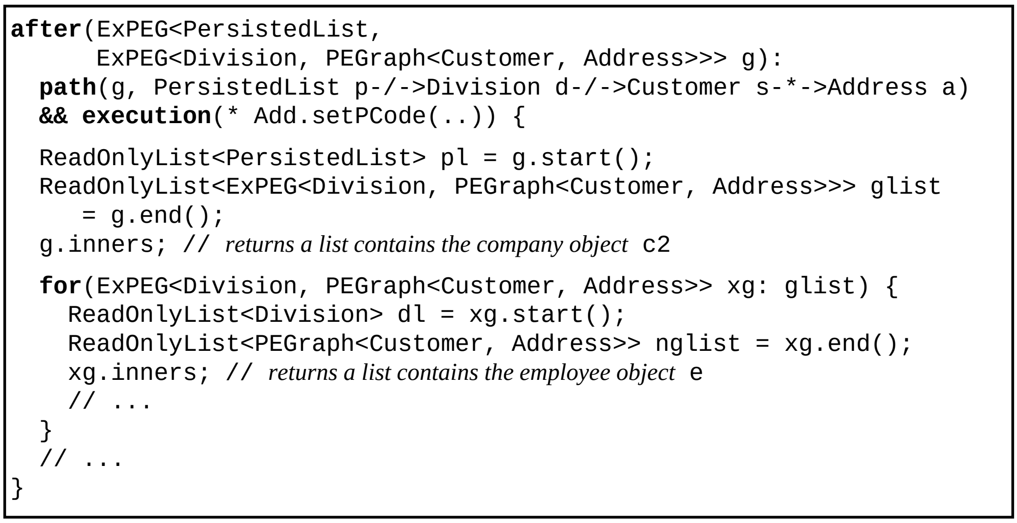
Figure 5.12: Path expression graph interface

For example, in the advice declaration of Figure 5.13, the type of the start objects is `PersistedList` and the type of the end objects is `Address`, which can be used directly without type-casting. Each owner `PersistedList` is notified after the change in the address object.

```
after(ExPEG<PersistedList,Address> g):
  execution(* Address.setPCode(*))
  && path(g, PersistedList p -/-> Address a)
{
  for(PersistedList p:g.start())
    p.notify(g.end().getFirst()); // ...
}
```

Figure 5.13: A simple example of `PEGraph` type

Similarly, situations with more complex path patterns can be handled as shown in Figure 5.14. Notice that the graph object in the list `nglist` does not have a list of inner objects.



```
after(ExPEG<PersistedList,
      ExPEG<Division, PEGraph<Customer, Address>>> g):
  path(g, PersistedList p-/->Division d-/->Customer s-*->Address a)
  && execution(* Add.setPCode(..)) {
  ReadOnlyList<PersistedList> pl = g.start();
  ReadOnlyList<ExPEG<Division, PEGraph<Customer, Address>>> glist
     = g.end();
  g.inners; // returns a list contains the company object c2
  for(ExPEG<Division, PEGraph<Customer, Address>> xg: glist) {
    ReadOnlyList<Division> dl = xg.start();
    ReadOnlyList<PEGraph<Customer, Address>> nglist = xg.end();
    xg.inners; // returns a list contains the employee object e
    // ...
  }
  // ...
}
```

Figure 5.14: A more complex example of `PEGraph` type

These examples show how to benefit from all type information included in the path pattern to avoid type-casting.

5.7.3 A Type System for PEP

In order to ensure that `PEGraph` types are correctly evaluated from the path patterns, a type system must be provided. This section proposes a simple type system to show how the PEP type is evaluated statically. This type system consists of a number of typing rules for the `PEGraph` type, its `start`, `end`, and `inners` list members. It does not consider how a complete pointcut definition is given a type, i.e. it considers only the type of the path pointcut.

A complete type system should cover dynamic typing of the term evaluation in any language. However, the intention of the thesis is to show how to ensure the type safety of the PEP construct that is required due to the use of the generic `PEGraph` type. Later on, one can integrate this type system with the available type systems for complete aspect-oriented languages, e.g. [Jagadeesan et al. (2006)]. Such integration requires slight refinements on some rules like the advice lookup and the ones that describe how a join point matches a pointcut. Moreover, the type system presented here does not consider the `PBList` type.

According to these assumptions, the evaluation typing is out of the scope of this section and consequently, there is no need to proof the preservation property of the type safety. The reason behind that is that the preservation property of the type system is part of the dynamic type evaluation [Pierce (2002)].

5.7.3.1 PEP Static Typing

The static typing for PEP is given for the pointcut that makes use of the path pointcut. The grammar of such pointcut definition is given in Table 5.3. Note that this syntax is exactly the same as the one defined at the beginning of this chapter with some abbreviated terms that are provided in order to ease the construction of the typing rules.

Table 5.3: Pointcut grammar

PC	$::=$	“$\texttt{pointcut}$” $pc(T_1\ v_1, \ldots T_n\ v_n)$: “$\texttt{path}$”$(VD,\ p)$
p	$::=$	$VD_1 \mathrel{-F\!\!\rightarrow} VD_2$
	$\mid$	$VD \mathrel{-F\!\!\rightarrow} p$
VD	$::=$	$T\ v \mid v$
v	$::=$	id
F	$::=$	“/” $\mid SF$
SF	$::=$	$id \mid$ “ $*$ ”
id	$::=$	*Defined according to the Java syntax*
T	$::=$	$Type \mid G\langle T_1, T_2\rangle \mid ExG\langle T_1, T_2\rangle$

Table 5.4 shows the static typing rules for PEP. These rules are divided into different categories. The first one, which is called [BoundVariables], consists of two rules that are responsible for collecting the bound variables of the given pointcut specifications. Rule [1.a] collects those variables and adds them to the global type environment Γ. As the rule shows, a bound variable is given according to the syntax of Table 5.3, e.g. $T\ v$, which then is added to the environment as $v : T$. This rule recursively calls itself until there is no more variables to be added. Then rule [1.b] minimizes the pointcut definition to PEP only.

Table 5.4: Static typing for PEP

[BoundVariables]

[1.a] $$\frac{\Gamma, \{v_1 : T_1\} \vdash \texttt{pointcut}\ pc(T_2\ v_2, \ldots T_n\ v_n) : \texttt{path}(VD,\ VD_1 -F\!\!\rightarrow p)}{\Gamma \vdash \texttt{pointcut}\ pc(T_1\ v_1, \ldots T_n\ v_n) : \texttt{path}(VD,\ VD_1 -F\!\!\rightarrow p)}$$

[1.b] $$\frac{\Gamma \vdash \texttt{path}(VD,\ VD_1 -F\!\!\rightarrow p)}{\Gamma \vdash \texttt{pointcut}\ pc() : \texttt{path}(VD,\ VD_1 -F\!\!\rightarrow p)}$$

[PEPatternAndPEGraph]

[2.a] $$\frac{\begin{array}{l}\Gamma \vdash g : G\langle T_1, T_2\rangle \\ \Gamma \uplus env(VD_1 -SF\!\!\rightarrow p) \vdash VD_1 : S_1 \\ \Gamma \uplus env(VD_1 -SF\!\!\rightarrow p) \vdash p : S_2 \\ \Gamma \vdash S_1 <: T_1,\ S_2 <: T_2\end{array}}{\Gamma \vdash \texttt{path}(g,\ VD_1 -SF\!\!\rightarrow p)}$$

[2.b] $$\frac{\begin{array}{l}\Gamma \uplus env(VD_1 -SF\!\!\rightarrow p) \vdash VD_1 : S_1 \\ \Gamma \uplus env(VD_1 -SF\!\!\rightarrow p) \vdash p : S_2 \\ \Gamma \vdash S_1 <: T_1,\ S_2 <: T_2\end{array}}{\Gamma \vdash \texttt{path}(G\langle T_1, T_2\rangle\ g,\ VD_1 -SF\!\!\rightarrow p)}$$

[3.a] $$\frac{\begin{array}{l}\Gamma \vdash g : ExG\langle T_1, T_2\rangle \\ \Gamma \uplus env(VD_1 -/\!\!\rightarrow p) \vdash VD_1 : S_1 \\ \Gamma \uplus env(VD_1 -/\!\!\rightarrow p) \vdash p : S_2 \\ \Gamma \vdash S_1 <: T_1,\ S_2 <: T_2\end{array}}{\Gamma \vdash \texttt{path}(g,\ VD_1 -/\!\!\rightarrow p)}$$

[3.b] $$\frac{\begin{array}{l}\Gamma \uplus env(VD_1 -/\!\!\rightarrow p) \vdash VD_1 : S_1 \\ \Gamma \uplus env(VD_1 -/\!\!\rightarrow p) \vdash p : S_2 \\ \Gamma \vdash S_1 <: T_1,\ S_2 <: T_2\end{array}}{\Gamma \vdash \texttt{path}(ExG\langle T_1, T_2\rangle\ g,\ VD_1 -/\!\!\rightarrow p)}$$

[4.a] $$\frac{\Gamma \vdash VD_1 : T_1 \qquad \Gamma \vdash p : T_2}{\Gamma \vdash VD_1 -SF\!\!\rightarrow p : G\langle T_1, T_2\rangle}$$

[4.b] $$\frac{\Gamma \vdash VD_1 : T_1 \qquad \Gamma \vdash p : T_2}{\Gamma \vdash VD_1 -/\!\!\rightarrow p : ExG\langle T_1, T_2\rangle}$$

[PEGraphComponents]

[5.a] $$\frac{\Gamma \vdash \texttt{path}(g : G\langle S, T\rangle,\ p)}{\Gamma \vdash g.\texttt{start} : List\langle S\rangle}$$

[5.b] $$\frac{\Gamma \vdash \texttt{path}(g : G\langle S, T\rangle,\ p)}{\Gamma \vdash g.\texttt{end} : List\langle T\rangle}$$

[5.c] $$\frac{\Gamma \vdash \texttt{path}(g : ExG\langle S, T\rangle,\ p)}{\Gamma \vdash g.\texttt{start} : List\langle S\rangle}$$

5.d] $$\frac{\Gamma \vdash \texttt{path}(g : ExG\langle S, T\rangle,\ p)}{\Gamma \vdash g.\texttt{end} : List\langle T\rangle}$$

[5.e] $$\frac{\Gamma \vdash \texttt{path}(g : ExG\langle S, T\rangle,\ p)}{\Gamma \vdash g.\texttt{inners} : List\langle \texttt{Object}\rangle}$$

The second category, which is called [PEPatternAndPEGraph], is responsible for the evaluation of the type of the path expression pattern, and then for checking whether the resulting type correctly corresponds to the type of the given PEGraph object. According to the grammar that is given in Table 5.3, a path pattern is either simple (represents a relation between two objects) or composed (represents a relation between more than two objects).

On the other hand, the relationship between two objects in the path pattern is either direct, which is denoted as SF, or indirect, which is represented by the wildcard "/". The simple path pattern of a direct relationship is of type G, while the one of an indirect relationship is of type ExG. The indirect relationship between two objects indicates that there is a list of inner objects in the resulting graph. Finally, the type of the resulting path expression graph is given either in the list of the bound variables of the pointcut header or it is included in the first parameter of the PEP. According to all of the above, there are eight different rules of this category.

In the type system of Table 5.4, four of these variations are considered whereas the other four rules are for the simple path patterns, which are very similar to the presented ones. For simplicity and readability reasons, those rules were omitted. Rules [2.a] and [2.b] check the well-formedness of the composed patterns with direct relationships (via the wildcard $*$ or a direct field name). Similarly, rules [3.a] and [3.b] are applicable for the path patterns with the indirect relationships.

According to rule [2.a], the PEP $\texttt{path}(VD,\ VD_1 \ {-SF}\rightarrow\ p)$, which is the result from the previous rules ([1.a] and [1.b]), is well-typed if all of the following holds. First, the type of the resulting graph is given in the pointcut header as $G\langle T_1, T_2\rangle\ g$, which means that g is typed inside Γ. Then, the current environment is concatenated with another typing context that represents the result from the auxiliary function $env()$. This concatenation is achieved by another auxiliary function that is denoted as $\uplus$, both of these auxiliary functions will be defined shortly. Finally, the resulting environment should entail that the type of the variable declaration VD is a subtype of T_1 and the type of the pattern p is a subtype of T_2. Rule [2.b] is similar to [2.a] except that the resulting graph type is given in the first PEP parameter.

Note that the subtyping relation is denoted as $<:$ after the conventions in [Cardelli (1997); Pierce (2002)]. Subtyping relation is transitive and reflexive.

Rules [3.a] and [3.b] are respectively similar to rules [2.a] and [2.b], however, they consider the typing for the path patterns with indirect relationships ("/"), where the resulting type is ExG.

The third category considers the types of the `PEGraph` components. The type of the `start` component of the $g : G\langle S, T\rangle$ is a generic list of type S (rules [5.a] and [5.c]), whereas the type of `end` component is a generic list of type T (rules [5.b] and [5.d]). The type $List\langle\texttt{Object}\rangle$ is used according to Pierce (2002). Moreover, rule [5.e] provides a typing for the inner objects in the resulting graph that is of type ExG.

The last set of rules represents the definitions of the auxiliary functions in Table 5.5. Rules [6.a] and [6.b] defines the recursive function $env(p)$. It takes a path pattern p and recursively minimizes it to collect the variable declarations and adds them to the environment Γ' by the help of the $\oplus$ function and the *typeDec* function. It terminates if there is no more variable declaration and returns the resulting environment to the caller.

Table 5.5: Auxiliary rules and functions of the PEP type system

$$\underline{[env(p)] \quad [typeDec(VD)] \quad [\Gamma \uplus \Gamma'] \quad [\Gamma \oplus typeDec(VD)]}$$

$$\text{[6.a]} \quad \frac{\{\} = \Gamma'}{env(VD) = \Gamma' \oplus typeDec(VD)} \qquad \text{[6.b]} \quad \frac{env(p) = \Gamma'}{env(VD_1 \ {-F\rightarrow}\ p) = \Gamma' \oplus typeDec(VD_1)}$$

$$\text{[7.a]} \quad typeDec(v) = \{\} \qquad \text{[7.b]} \quad typeDec(T\ v) = \{v : T\}$$

$$\text{[8.a]} \quad \frac{}{\Gamma' \uplus \{\}\ = \Gamma'} \qquad \text{[8.b]} \quad \frac{(\Gamma' \oplus \{v_1 : T_1\}) \uplus \{v_2 : T_2, \ldots, v_n : T_n\}}{\Gamma' \uplus \{v_1 : T_1, \ldots, v_n : T_n\}}$$

$$\text{[9.a]} \quad \frac{}{\Gamma' \oplus \{\}\ =\ \Gamma'} \qquad \text{[9.b]} \quad \frac{\Gamma' \vdash v : T}{\Gamma' \oplus \{v : T\}\ =\ \Gamma'} \qquad \text{[9.c]} \quad \frac{(v : T) \notin \Gamma'}{\Gamma' \oplus \{v : T\}\ =\ \Gamma', \{v : T\}}$$

$$\text{[9.d]} \quad \frac{\Gamma' \vdash v : T_1 \quad \vdash T_1 <: T_2}{\Gamma' \oplus \{v : T_2\}\ = \Gamma'} \qquad \text{[9.e]} \quad \frac{\Gamma' \vdash v : T_1 \quad \vdash T_2 <: T_1}{\Gamma' \oplus \{v : T_2\}\ = (\Gamma' - \{v : T_1\}), \{v : T_2\}}$$

$$\text{[9.f]} \quad \Gamma' \oplus \{v : T\} = \{v : T\} \oplus \Gamma'$$

$$\text{[10.a]} \quad \frac{\Gamma \vdash v : T}{\Gamma \vdash T\ v : T} \qquad \text{[10.b]} \quad \frac{v : T \in \Gamma}{\Gamma \vdash v : T} \qquad \text{[10.c]} \quad \frac{\Gamma \vdash v : T_2 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash T_1\ v : T_2}$$

Rules [7.a] and [7.b] define the *typeDec* function. It returns an empty environment $\{\}$, if the variable declaration *VD* is a simple variable v. If *VD* is of the form $T\ v$, the function returns an a type environment that contains only one mapping, i.e. $\{v : T\}$.

The environment concatenation binary operator $\uplus$ is defined in rules [8.a] and [8.b]. The later reduces the environment on the right operand of the operator while adding each type information to the environment Γ'. Rule [8.a] is the stop criteria of the operation.

Rules [9.a] to [9.f] combine a single type binding with an existing environment by using the $\oplus$ operand. This combination is defined to choose the least common subtype for the variable that is already bound in the target environment. This meets the type resolution process that is discussed in the last chapter. If the term v is already bound to the type T_1 in the environment Γ' where T_1 is a supertype of T_2, the combination of Γ' and $\{v : T_2\}$ implies that the type of v is of the subtype T_2. The last rule indicates that the $\oplus$ operation is symmetric.

Finally, the variable declaration $T\ v$ is of type T is given by rule [10.a], while rule [10.b] is the simple variable type rule, which is know as `[T-Var]` in [Pierce (2002)].

5.7.3.2 Examples

The first example in Table 5.6 shows a type derivation tree for a simple path pointcut. This derivation explains how the type system is working. Note that the subtyping relation in the usage of rule [2.a] in the derivation is omitted due to the space restriction, however, it is clear that $A <: A$ and $B <: B$ from the reflexive property of the subtyping relationship. For the same reason, the environment of the right most part of the tree is denoted as "..." since it is exactly the same environment that is evaluated in the middle part of the tree.

Table 5.7 elaborates another example for more complex path expression pattern. The resulting graph is of type $G\langle A, ExG\langle B, C\rangle\rangle$, which reflects the type of the composed pattern $a \ {-}{*}{\rightarrow}\ B2\ b\ {-}/{\rightarrow}\ C\ c$. Note that the type of the `PEGraph` object is given inside PEP, which means that it is not part of the bound variables.

In this example, the abbreviation assumptions that are used in the previous example are used. The subtyping relationship is applied in this example since $B2$

is a subtype of B. The evaluation of the auxiliary functions is shown in detail only for the case of $B2\ b$, and the other cases are similar.

The last example in Table 5.8 shows how the type system detects the type errors in a given path expression pointcut. The resulting type environment is not well-typed, since according to the given `PEGraph` type, variable b is expected to be of type C. The type system raises a type error when it find that the declared type of b in the path pattern is B and not C (type B is not a subtype of C).

5.8 Chapter Summary

As path expression pointcut involves a number of complex concepts and structures, a formalization of it was derived step-by-step in this chapter. For the purpose of self-containedness, a brief introduction to the denotational formal semantics was provided by using an illustrative example for the semantics of a simple calculator.

It was necessary to build a mathematical base model for the context that PEP applied in, i.e. the object graphs. Then, the abstract syntax or PEP was specified to show the syntactic phrases that requires semantics interpretations by means of semantic functions. The semantic domains, functions, and equations have been specified for PEP.

Understanding PEP in the context of a complete pointcut language requires a specification of the meaning of complete pointcut definitions that make use of PEP. This is necessary for any implementation of PEP in any aspect-oriented language as well as for the usage of PEP. Accordingly, the formal semantics of PEP was integrated with the semantics of Aspect SoundBox.

Afterwards, two generic types for the path expression graph and the binding list were provided along with a simple type system that explains how correct types for the variables within the path pointcut are evaluated.

Table 5.6: Type derivation example 1

```
pointcut pc(PEGraph<A, B> g, A a):  path(g, a -*-> B b);
```

$$\phi \vdash \texttt{pointcut}\ \ pc(G\langle A,B\rangle\ g,\ A\ a) : \texttt{path}(g, a -\!\ast\!\rightarrow B\ b)$$

$$\phi, \{g : G\langle A,B\rangle\} \vdash \texttt{pointcut}\ \ pc(A\ a) : \texttt{path}(g,\ a\ -\!\ast\!\rightarrow B\ b) \qquad [1.a]$$

$$\phi, \{g : G\langle A,B\rangle, a : A\} \vdash \texttt{pointcut}\ \ pc() : \texttt{path}(g,\ a\ -\!\ast\!\rightarrow B\ b) \qquad [1.a]$$

$$\phi, \{g : G\langle A,B\rangle, a : A\} \vdash \texttt{path}(g,\ a\ -\!\ast\!\rightarrow B\ b) \qquad [1.b]$$

$\phi, \{g{:}G\langle A,B\rangle, a{:}A\} \vdash g{:}G\langle A,B\rangle$		$\phi, \{g{:}G\langle A,B\rangle, a{:}A\} \uplus env(a -\!\ast\!\rightarrow B\ b) \vdash a{:}A$		$\phi, \{g{:}G\langle A,B\rangle, a{:}A\} \uplus env(a -\!\ast\!\rightarrow B\ b) \vdash B\ b{:}B$	[2.a]
$g{:}G\langle A,B\rangle \in \phi, \{g{:}G\langle A,B\rangle, a{:}A\}$	[10.b]	$\phi, \{\dots\} \uplus (env(B\ b) \oplus typeDec(a)) \vdash a{:}A$	[6.b]	$\cdots \vdash B\ b{:}B$	[6.b]
$\phi, \{g{:}G\langle A,B\rangle, a{:}A\} \vdash \diamond$		$\phi, \{\dots\} \uplus (\{\} \oplus typeDec(B\ b) \oplus typeDec(a)) \vdash a{:}A$	[6.a]	$\cdots \vdash B\ b{:}B$	[6.a]
		$\phi, \{\dots\} \uplus (\{\} \oplus \{b{:}B\} \oplus typeDec(a)) \vdash a{:}A$	[7.b]	$\cdots \vdash B\ b{:}B$	[7.b]
		$\phi, \{\dots\} \uplus (\{b{:}B\} \oplus typeDec(a)) \vdash a{:}A$	[9.a]	$\cdots \vdash B\ b{:}B$	[9.a]
		$\phi, \{\dots\} \uplus (\{b{:}B\} \oplus \{\}) \vdash a{:}A$	[7.a]	$\cdots \vdash B\ b{:}B$	[7.a]
		$\phi, \{g{:}G\langle A,B\rangle, a{:}A\} \uplus \{b{:}B\} \vdash a{:}A$	[9.a]	$\cdots \vdash B\ b{:}B$	[9.a]
		$(\phi, \{g{:}G\langle A,B\rangle, a{:}A\} \oplus \{b{:}B\}) \uplus \{\} \vdash a{:}A$	[8.b]	$\cdots \vdash B\ b{:}B$	[8.b]
		$\phi, \{g{:}G\langle A,B\rangle, a{:}A, b{:}B\} \uplus \{\} \vdash a{:}A$	[9.c]	$\cdots \vdash B\ b{:}B$	[9.c]
		$\phi, \{g{:}G\langle A,B\rangle, a{:}A, b{:}B\} \vdash a{:}A$	[9.a]	$\phi, \{g{:}G\langle A,B\rangle, a{:}A, b{:}B\} \vdash B\ b{:}B$	[9.a]
		$a{:}A \in \phi, \{g{:}G\langle A,B\rangle, a{:}A, b{:}B\}$	[10.b]	$\phi, \{g{:}G\langle A,B\rangle, a{:}A, b{:}B\} \vdash B\ b{:}B$	[9.a]
		$\phi, \{g{:}G\langle A,B\rangle, a{:}A, b{:}B\} \vdash \diamond$		$\phi, \{g{:}G\langle A,B\rangle, a{:}A, b{:}B\} \vdash b{:}B$	[10.a]
				$b{:}B \in \phi, \{g{:}G\langle A,B\rangle, a{:}A, b{:}B\}$	[10.b]
				$\phi, \{g{:}G\langle A,B\rangle, a{:}A, b{:}B\} \vdash \diamond$	

Table 5.7: Type derivation example 2

```
pointcut pc(A a, B b):  path(PEGraph<A, ExPEG<B, C>> g, a -*-> B2 b -/-> C c); // B2 extends B
```

		$\phi \vdash \texttt{pointcut}\ \ pc(A\ a, B\ b) : \texttt{path}(G\langle A, ExG\langle B, C\rangle\rangle\ g, a \relbar\!*\!\!\rightarrow B2\ b \relbar\!/\!\!\rightarrow C\ c)$			
		$\phi, \{a\!:\!A\} \vdash \texttt{pointcut}\ \ pc(B\ b) : \texttt{path}(G\langle A, ExG\langle B, C\rangle\rangle\ g, a \relbar\!*\!\!\rightarrow B2\ b \relbar\!/\!\!\rightarrow C\ c)$			[1.a]
		$\phi, \{a\!:\!A\}, \{b\!:\,B\} \vdash \texttt{pointcut}\ \ pc() : \texttt{path}(G\langle A, ExG\langle B, C\rangle\rangle\ g, a \relbar\!*\!\!\rightarrow B2\ b \relbar\!/\!\!\rightarrow C\ c)$			[1.a]
		$\phi, \{a\!:\!A, b\!:\!B\} \vdash \texttt{pointcut}\ \ pc() : \texttt{path}(G\langle A, ExG\langle B, C\rangle\rangle\ g, a \relbar\!*\!\!\rightarrow B2\ b \relbar\!/\!\!\rightarrow C\ c)$			[1.a]
		$\phi, \{a\!:\!A, b\!:\!B\} \vdash \texttt{path}(G\langle A, ExG\langle B, C\rangle\rangle\ g, a \relbar\!*\!\!\rightarrow B2\ b \relbar\!/\!\!\rightarrow C\ c)$			[1.b]
$\cdots \vdash a\!:\!A$		$\phi, \{a\!:\!A, b\!:\!B\} \uplus env(a \relbar\!*\!\!\rightarrow B2\ b \relbar\!/\!\!\rightarrow C\ c) \vdash B2\ b \relbar\!/\!\!\rightarrow C\ c\!:\!ExG\langle B, C\rangle$			[2.b]
$\cdots \vdash a\!:\!A$	[6.b]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus env(a \relbar\!*\!\!\rightarrow B2\ b \relbar\!/\!\!\rightarrow C\ c) \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[4.b]
$\cdots \vdash a\!:\!A$	[6.b]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus (env(B2\ b \relbar\!/\!\!\rightarrow C\ c) \oplus typeDec(a)) \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[6.b]
$\cdots \vdash a\!:\!A$	[6.a]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus (env(C\ c) \oplus typeDec(B2\ b) \oplus typeDec(a)) \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[6.b]
$\cdots \vdash a\!:\!A$	[7.b]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus (\{\} \oplus typeDec(C\ c) \oplus typeDec(B2\ b) \oplus typeDec(a)) \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[6.a]
$\cdots \vdash a\!:\!A$	[9.a]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus (\{\} \oplus \{c\!:\!C\} \oplus typeDec(B2\ b) \oplus typeDec(a)) \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[7.b]
$\cdots \vdash a\!:\!A$	[7.b]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus (\{c\!:\!C\} \oplus typeDec(B2\ b) \oplus typeDec(a)) \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[9.a]
$\cdots \vdash a\!:\!A$	[9.c]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus (\{c\!:\!C\} \oplus \{b\!:\!B2\} \oplus typeDec(a)) \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[7.b]
$\cdots \vdash a\!:\!A$	[7.b]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus (\{c\!:\!C, b\!:\!B2\} \oplus typeDec(a)) \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[9.c]
$\cdots \vdash a\!:\!A$	[9.a]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus (\{c\!:\!C, b\!:\!B2\} \oplus \{\}) \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[7.b]
$\cdots \vdash a\!:\!A$	[8.b]	$\phi, \{a\!:\!A, b\!:\!B\} \uplus \{c\!:\!C, b\!:\!B2\} \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[9.a]
$\cdots \vdash a\!:\!A$	[9.c]	$(\phi, \{a\!:\!A, b\!:\!B\} \oplus \{c\!:\!C\}) \uplus \{b\!:\!B2\} \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[8.b]
$\cdots \vdash a\!:\!A$	[8.b]	$(\phi, \{a\!:\!A, b\!:\!B\ c\!:\!C\}) \uplus \{b\!:\!B2\} \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[9.c]
$\cdots \vdash a\!:\!A$	[9.e]	$(\phi, \{a\!:\!A, b\!:\!B\ c\!:\!C\} \oplus \{b\!:\!B2\}) \uplus \{\} \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[8.b]
$\phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\} \vdash a\!:\!A$	[9.a]	$(\phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\}) \uplus \{\} \vdash B2\ b\!:\!B$		$\cdots \vdash C\ c\!:\!C$	[9.e]
$a\!:\!A \in \phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\}$	[10.b]	$\phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\} \vdash B2\ b\!:\!B$		$\phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\} \vdash C\ c\!:\!C$	[9.a]
$\phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\} \vdash \diamond$		$\phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\} \vdash b\!:\!B2$	[10.c]	$\phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\} \vdash c\!:\!C$	[10.a]
		$b\!:\!B2 \in \phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\}$	[10.b]	$c\!:\!C \in \phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\}$	[10.b]
		$\phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\} \vdash \diamond$		$\phi, \{a\!:\!A, c\!:\!C, b\!:\!B2\} \vdash \diamond$	

Table 5.8: Type derivation example 3

```
pointcut pc():  path(PEGraph<A, C> g, A a -*-> B b);
```

$$\phi \vdash \texttt{pointcut}\ \ pc() : \texttt{path}(G\langle A, C\rangle\ g, A\ a \mathrel{-\!\!*\!\!\longrightarrow} B\ b)$$

$\phi \vdash \texttt{path}(G\langle A, C\rangle\ g, A\ a \mathrel{-\!\!*\!\!\longrightarrow} B\ b)$			[1.b]
$\phi \uplus env(A\ a \mathrel{-\!\!*\!\!\longrightarrow} B\ b) \vdash A\ a\!:\!A$		$\phi \uplus env(A\ a \mathrel{-\!\!*\!\!\longrightarrow} B\ b) \vdash B\ b\!:\!C$	[2.b]
$\cdots \vdash A\ a\!:\!A$		$\phi \uplus (env(B\ b) \oplus typeDec(A\ a)) \vdash B\ b\!:\!C$	[6.b]
$\cdots \vdash A\ a\!:\!A$		$\phi \uplus (\{\} \oplus typeDec(B\ b) \oplus typeDec(A\ a)) \vdash B\ b\!:\!C$	[6.b]
$\cdots \vdash A\ a\!:\!A$		$\phi \uplus (typeDec(B\ b) \oplus typeDec(A\ a)) \vdash B\ b\!:\!C$	[9.a]
$\cdots \vdash A\ a\!:\!A$		$\phi \uplus (\{b\!:\!B\} \oplus \{a\!:\!A\}) \vdash B\ b\!:\!C$	[7.b]
$\cdots \vdash A\ a\!:\!A$		$\phi \uplus \{b\!:\!B, a\!:\!A\} \vdash B\ b\!:\!C$	[9.c]
$\cdots \vdash A\ a\!:\!A$		$(\phi \oplus \{b\!:\!B\}) \uplus \{a\!:\!A\} \vdash B\ b\!:\!C$	[8.b]
$\cdots \vdash A\ a\!:\!A$		$\phi, \{b\!:\!B\} \uplus \{a\!:\!A\} \vdash B\ b\!:\!C$	[9.c]
$\cdots \vdash A\ a\!:\!A$		$(\phi, \{b\!:\!B\} \oplus \{a\!:\!A\}) \uplus \{\} \vdash B\ b\!:\!C$	[8.b]
$\cdots \vdash A\ a\!:\!A$		$\phi, \{b\!:\!B, a\!:\!A\} \uplus \{\} \vdash B\ b\!:\!C$	[9.c]
$\phi, \{b\!:\!B, a\!:\!A\} \vdash a\!:\!A$	[10.a]	$\phi, \{b\!:\!B, a\!:\!A\} \vdash B\ b\!:\!C$	[8.a]
$a\!:\!A \in \phi, \{b\!:\!B, a\!:\!A\}$	[10.b]	$B \not<: C \Rightarrow B\ b\!:\!C \notin \phi, \{b\!:\!B, a\!:\!A\}$	
$\phi, \{b\!:\!B, a\!:\!A\} \vdash \diamond$		$\phi, \{b\!:\!B, a\!:\!A\} \nvdash \diamond$	

Chapter 6

Implementation

Whereas the previous two chapters discussed the conceptual issues of path expression pointcuts, this chapter discusses the practical part of the thesis. A prototype implementation of path pointcut in a small aspect-oriented language is introduced below. The resulting language is called aspect language with path expression pointcut (PePAL).

6.1 Aspect Language with Path Expression Pointcut (PePAL)

PePAL is a minimal typed aspect-oriented programming language that defines path expression pointcut (PEP) on top of a join point model that contains only set operation join points. The intention is to show how PEP is working as a binder pointcut along with *kinded* pointcuts [Hilsdale and Hugunin (2004)] that designate corresponding join points, which in PePAL are field set join points.

In order to study how PEP interacts with other binder pointcuts, PePAL's `set` pointcut also binds the *target* object of the set operation. In addition to the target object, `set` pointcut also maintains the *argument* object or value, which represents the value the field is being set to. This occurs in fact to reflect the semantics specified in the last chapter.

For the purpose of readability and comprehension, PePAL has the same syntactic format of AspectJ language, since the later is the most prominent aspect-oriented programming language.

The context of PEP is the object graph, hence, the virtual machine of PePAL is built in such a way as to provide a suitable data structure that holds the object graph. Moreover, it provides an efficient manipulation of the object graph. Whenever a given PEP is to be evaluated, the object graph is passed to the aspect that can reason about it. A suitable representation of the object graph matches directly the mathematical model presented in the previous chapter, which is shown to define precise semantics for PEP [Al-Mansari et al. (2008)].

Reflecting the formal semantics and being a prototype implementation, PePAL does not consider the garbage collection.

These restrictions on the implementation are considered in order to prove the feasibility of the PEP construct. The limited pointcut language of PePAL is enough to solve the problems described in Chapter 3.

In what follows, the syntax of the language is explained along with some illustrative examples of its usage.

6.1.1 Syntax

PePAL's compilation unit consists of a number of classes. A class, as in Java, can have method or field members. Each class is a subclass of the general superclass `Object`. If a given class contains an advice declaration, this class is considered as an aspect. PePAL's aspect members are only pieces of advice, and nothing else. For example, PePAL does not support inter-type declarations and `declare` clauses.

PePAL supports single inheritance and does not provide means for interfaces. Since aspects are classes by nature, the language permits inheritance between aspects. The classes and aspects are always concrete, and there is no mean to define abstract classes and members. The language also eliminates the modifiers.

The PePAL BNF grammar is as follows:

PePAL GRAMMAR

```
compilationUnit ::= (class)*
class ::=‘‘class’’ id [typePars] [‘‘extends’’ id]‘‘{’’(cMember)*‘‘}’’
```

```
cMember ::= advice | field | method
advice ::= type ``around'' ``('' [formals] ``)'': pcBody aBlock
field ::= type id ``;''
method ::= rType [``static''] id ``('' [formals] ``)'' mBlock
type ::= boolean | int | id | id [typePars] | String | Object
rType ::= ``void'' | type
typePars ::= ``<'' id (``,'' id)* ``>''
formals ::= type id (``,'' type id)*
aBody ::= ``{'' (block)* [proceedStmt] (block)* ``}''
block ::= variable ``;'' | stmt
proceedStmt ::= ``proceed'' ``('' [ args ] ``)'' ``;''
mBlock ::= ``{'' (block)* ``}''
variable ::= type id
stmt ::= ``if'' ``('' expression ``)'' mBlock ``else'' mBlock
       | ``for'' ``('' id id ``:'' id ``)'' mBlock
       | [ expStm ] ``;''
       | ``throw'' ``exception'' ``;''
       | ``return'' [ expression ] ``;''
expStm ::= methodCall | assignment
methodCall ::= expression ``.'' id ``('' [ args ] ``)''
             | id ``('' [ args ] ``)''
args ::= expression (``,'' expression )*
assignment ::= id ``='' expression
expression ::= methodCall
             | assignment
             | id
             | integer | string
             | ``true'' | ``false'' | ``this''
             | ``new'' id ``('' ``)''
             | ``('' expression ``)''

pcBody ::= pcOr
pcOr ::= pcAnd | pcOr ``||'' pcAnd
pcAnd ::= aPC | pcAnd ``&&'' aPC | ``('' pcOr ``)''
aPC ::= setPC | pathPC
setPC ::=``set'' ``('' id id ``.'' (id | ``*'') ``,''  id ``,'' id ``)''
```

```
pathPC ::= ‘‘path’’ ‘‘(’’ id ‘‘,’’ pathExprPattern ‘‘)’’
pathExprPattern ::= id ( ‘‘-’’ (id | ‘‘*’’ | ‘‘/’’) ‘‘->’’ id)+
```

Types in PePAL are either primitive types, `int` and `boolean`, or reference types that may be parameterized, and can only be class types since there is no support for interfaces in PePAL. `String` is a built-in type in PePAL.

The available statements in PePAL are conditional `if`-statements, simple `throw` clauses, `return`, method invocations, and assignments. Similarly, some expression variations are supported like variable read, values of integer, boolean and string, the `this` operator, and a default constructor.

Only `around` advice is provided with proceeding functionality by using the keyword `proceed`. The pointcuts in PePAL are anonymous, i.e. they are defined directly in the advice declarations.

The pointcut language of PePAL contains only the `path` and the `set` pointcuts. These primitive pointcut designators can be grouped conjunctively, by means of "`&&`", or disjunctively, by means of "`||`", to compose more complex pointcut definitions. PePAL eliminates the usage of negation "`!`" for pointcuts. As stated above, the target object and the argument of a set join point is exposed by the `set` pointcut itself.

Finally, the syntax shows that PePAL provides a very strict syntax for PEP to the one that has been used in the last two chapters. The only difference is that PePAL eliminates the usage of type patterns and simply uses exact names for types.

6.1.2 Examples

Classes in PePAL are defined exactly as in Java. Aspects are classes that have advice declarations. For example, the code in Figure 6.1 declares a valid class `Person` and a valid aspect `AddressMonitor` that selects all set operations on the address field of person objects. After proceeding the execution of the assignment, the advice writes these changes to the log profile.

According to `AddressMonitor` aspect, the first assignment statement (setting the street property) will not be selected since the field `street` is of `String` type,

```
class Address {
  String street;
  int number;
  int postCode;
  String city;
  // ...
}

class Person {
  String name;
  Address residence;
  // ...
}

class AddressMonitor {
  // ...
  Object around(Person p, Address a): set(Address Object.*, p, a){
    Object ret = proceed(p, a);
    this.writeLog(p, a);
    return ret;
  }
}

class Main {
  void main() {
    Person per = new Person();
    Address addr = new Address();
    // assign values to addr1 properties
    addr.street = ``Essenerstr.´´;
    // ...
    per.residence = addr;
    // ...
  }
}
```

Figure 6.1: Example of declaring classes and aspects in PePAL

while the `set` pointcut selects only the field set operations if the field is of type `Address`. However, the last assignment statement will be selected, and the resulting binding set will be $\{(p, per), (a, addr)\}$, where per is the target object of the set operation and $addr$ is the argument value that represents the new value of the modified field.

Figure 6.2 shows a possible PePAL implementation of the parts of the company object model shown in Figure 3.1 in Chapter 3.

```
class Company {
  List<Employee> employees;
  List<Customer> customers;
  // ...
}

class Customer extends Person {
  Address billTo;
  Address shipTo;
  int balance;
  // ...
}

class Employee extends Person {
  String ssn;
  int salary;
  // ...
}
```

Figure 6.2: Example of declaring a class hierarchy in PePAL

Figure 6.3 shows an example of how `path` pointcut is used. The example uses the `PEGraph` API that is added to the language as described in the last section.

```
class AddressMonitor {
  // ...
  Object around(PEGraph gid, Company c, Address a):
    set(Object Address.*, a, Object) && path(gid, c -/-> a)
  {
    Object ret = proceed(c, a);
    writeLog(c, a);
    return ret;
  }
}
```

Figure 6.3: Example of using PEP in PePAL

Notice that, here, the company object that owns the changed address object is being exposed to the advice, so there is no need to obtain it from the exposed path graph. That is why the used `PEGraph` type is not parameterized. Only assignment statements on the address objects that are reachable from a given company object are selected.

In this example, notice also that the general superclass `Object` is used in `set` pointcut as a return type and also as a type for the argument value of the set operation. This is the case when the developers are not interested in the return type and in exposing the target object and/or the argument. It is a substitution of the wildcard "`*`".

The code example in Figure 6.4 shows the main method of the application that instantiates objects and performs some field assignments to them, which might be potentially selected by the `AddressMonitor` aspect.

```
class Main {
  void static main() {
    Address a1 = new Address();
    Company c1 = new Company();
    Customer s1 = new Customer();
    a1.street = ...; // not selected since a1 is part of no company
    s1.billTo = a1;
    s1.residence = a1;
    c1.customers.add(s1)
    a1.street = ...; // selected
  }
}
```

Figure 6.4: Example of potential join points in the main method in PePAL

As mentioned in Chapter 4, if the developer needs to get access to all company objects that own the updated address objects, the company object must not be part of the advice header (must not be a bound object). This is can be achieved in PePAL as shown in Figure 6.5.

Whenever a field assignment is executed on an address object that is reachable from one or more company objects, the advice will fire and will invoke method `writeLogMultiple`. This method has two arguments, the first is the list of the start nodes in the resulting `gid` and the second is the changed object, and it will iterate through all start objects to perform the required logging operation. Notice that the bindings are for variables `gid` and `a`.

```
class AddressMonitor {
  // ...
  Object around(ExPGT<Company, Address> gid, Address a):
    set(Object Address.*, a, *) && path(gid, Company c -/-> a)
  {
    Object ret = proceed(gid, a);
    writeLogMultiple(gid.start(), a);
    return ret;
  }
  // ...
  void writeLogMultiple(List<Company> l, Address a) {
    writeLog(l.removeFirst(), a);
    if(l.notEmpty())
      writeLogMultiple(l, a);
  }
}
```

Figure 6.5: Example of exposing all owner company objects of a shared address object

The final code example that is shown in Figure 6.6 shows that, if the developers are not interested in exposing the `PEGraph` object to the advice, they should not define it in the formal parameters of the advice and simply define it in the PEP itself:

```
class AddressMonitor {
  // ...
  Object around(Company c, Address a):
    set(Object Address.*, a, *) && path(PEGraph gid, c -/-> a)
  {
    Object ret = proceed(c, a);
    writeLog(c, a);
    return ret;
  }
}
```

Figure 6.6: The `PEGraph` object is not exposed to the advice

6.2 Design Issues

According to the graph model that is presented in the last chapter, objects graphs are directed and usually contain cycles. This is considered as a source of complexity since there might be an infinite number of matching paths to a given path expression pattern in such structure in the presence of the "/" wildcard.

In PePAL, two restrictions have been introduced in order to minimize this complexity and to guarantee the termination of the path evaluation process. First, the set of bound variables -corresponds to BV in the semantics- is used to control the process of finding the matching paths in the object graph. Whenever a cyclic path is found to cover all needed bindings, the traverse process stops and considers this path as a matching path, then the process continues to find other paths. Second, each cycle should not be visited more than once unless the first restriction has not been already fulfilled.

For example, consider the following advice declaration:

```
void around(A a, C c, D d): path(PEGraph g, a -/-> c -/-> d) ...
```

According to the object collaboration diagram in Figure 6.7, there exists infinite number of paths that are matching the given path pattern, because of the presence of the cycle between objects `o2`, `o3`, and `o4`.

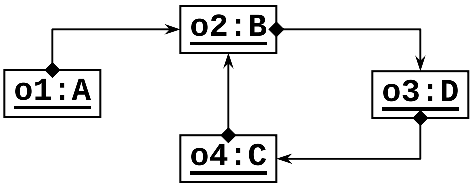


Figure 6.7: Cyclic object graph

Assume that the traversal process starts from object `o1` that will be bound to variable `a`, and then it visits `o2`, `o3` and `o4`. Till this point, the traversal satisfies the first part of the path pattern, i.e. `a -/-> c`, with the binding (`a=o1, c=o4`) but still there is no binding for the variable `d`. This means that the first restriction is still not fulfilled. The traversal continues to visit the object `o2` again, which means that a cycle has been detected. The only way to proceed is to go through the cycle again where the rest of the required bindings must be found. Then the algorithm visits object `o3` again to bind it to the variable `d`.

6.3 Chapter Summary

This chapter presents a minimal aspect-oriented programming language with path expression pointcut (PePAL) that only supports the field set join points. The description of PePAL includes its features as a typed object-based language. Then, the syntax of the language was introduced along with code examples to illustrate its usage. This short description is sufficient to understand the solutions presented in the next chapter for the thesis motivating examples.

Chapter 7

Motivating Examples Revisited

This chapter shows how PePAL can be used to solve the problems discussed in Chapter 3. The first example shows how persisting containers and PEP are used to designate the join points where updates and retrieve operations are issued on persistent objects. The second example will discussed how an aspectized field-based locking mechanism is implemented elegantly in PePAL. Then, an aspect that solves the cascading version locking policy in PePAL will be provided. The last example gives a solution for the problem of observer design pattern that was discussed in Chapter 3.

7.1 Pure Persistence Problem: Persisting Updates

In Section 3.3.1, Figure 3.21 presented a workaround-solution for the object reachability problem that current aspect-oriented persistence solutions suffer from. The problem was originated from the generalized quantification in these systems.

Recall the definition of `trapUpdates` in `ApplicationDatabaseAccess` aspect:

```
pointcut trapUpdates(Object o):
set(* *.*) && target(o) && if(reachable(pList, o));
```

This pointcut should pick up every field set join point in any object that is reachable from the persistent list `pList`. This relationship was obtained by means of hand-coded routines that conceptually do not reflect the semantics of join point selection by means of pointcuts, in addition to their lack of efficient performance.

Using `path` pointcut, an elegant conceptually fit solution to the semantics of join point selection can be provided without the need to force the developer to hand-code sophisticated procedures by using complex reflective facilities.

```
class ApplicationDatabaseAccess {
  Object around(PersistedList p, Object o):
    set(Object Object.*, o, Object) && path(PEGraph gid, p -/-> o)
  {
    Object ret = proceed(p, o);
    p.persist(o);
    return ret;
  }
}
```

Figure 7.1: Using PEP to make object updates persistent

Figure 7.1 shows a PePAL code that provides such solution. The path pattern in this PEP provides better abstraction over the reachability relationship between the objects. Moreover, this pointcut is robust against any changes to the class hierarchy of the application.

Notice that the use of persistent list and PEP complies with the type orthogonality principle of orthogonal persistence. The above solution does not require the types to be prepared for persistence, since all objects of any type are persistent as long as they are reachable from a persistent list.

7.2 Pessimistic Field-based Locking Policy

Recall the problem that is described in Section 3.3.2. Figure 7.2 depicts the problem collaboration diagram again. A complex workaround solution was provided in Figure 3.23 and Figure 3.24. It was shown how this solution suffers from practical and conceptual serious problems. The advice has to participate in selecting the join point due to the absence of good pointcut abstraction over the non-local field information.

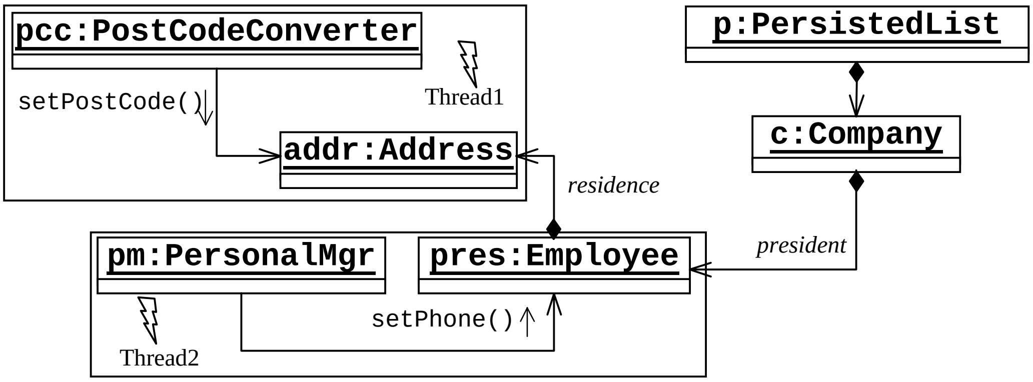


Figure 7.2: Concurrent threads and the problem of non-local field information

Using PEP and persisting containers, one can recover these practical and conceptual problems easily. As part of its ad-hoc functionality, each persistent list acquires locks for the fields that are part of it and caches all these locks. The first functionality is achieved by invoking method `acquireLock(f)` on the persistent list, where `f` is the field to be locked. In order to check whether a given field is already locked, one can use method `isLocked(f)` of the persistent list interface.

Consider the PePAL code that is shown in Figure 7.3. The locking is applied on the fields of any object `o1` of type `T` (as shown in the path pattern). The application of `g.end()` returns an object of type `PEGraph<T, Object>` whose `nextFields(t)` method returns the fields of object `t` that are need to be locked. One can use `Employee` instead of `T` to solve the case of the motivating example in Section 3.3.2.

Whenever an object `o2` that belongs to `T` object(s) is changed, the `around` advice tries to lock all fields of the `T` object(s) by which `o2` is referenced. If the field is already locked, the advice raises an exception, otherwise, the field is acquired a lock.

```
class FLConcurrencyControlAspect {
  Object around(PEGraph<PersistedList, PEGraph<T, Object>> g,
                Object o2):
    set(Object Object.*, o2, Object) &&
    path(g, PersistedList p -/-> T o1 -/-> o2)
  {
    List visited = new List();
    for(PersistedList p : g.start()) {
      for(T t : g.end().start()) {
        if(visited.notContains(t)) {
          visited.add(t);
          // get the dirty fields of t
          for(Field df : g.end().nextFields(t)) {
            if(p.isLocked(df)) {
              throw exception;
            } else { p.acquireLock(df); }
          }
        }
      }
    }
    Object ret = proceed(g, o2);
    return ret;
  }
}
```

Figure 7.3: Field-based locking policy in PePAL

7.3 Optimistic Version-based Locking Policy with Cascading

The problem of getting access to all owner objects of the changed persistent object was described in Section 3.3.3. Before explaining how version-based locking policy can be implemented by using the persistent lists and PEP, a number of assumptions must be explained. This thesis considers using numeric versions only.[1] The aspect maintains a global cache GCache that maps each object to its version from the time it is read first from the data store until its deletion. All versions are initially set to 1 and they are updated whenever their corresponding objects are saved to the data store.

[1] Other version locking solutions may use timestamps instead of numeric version values [Elmasri and Navathe (2000)].

As a part of the ad-hoc functionality of the persistent lists that is similar to the *spontaneous containers* [Popovici et al. (2003)], it is assumed that each persistent list maintains a list of the form `(object, version)`. Such element is added to the list whenever a thread reads an object of the persistent list with the current version of that object from `GCache`. Moreover, it is assumed that the persistent lists provide the method `persist(o)` to save object `o` to the data store.

```
class CascadingVLPolicy {
  GCache gCache = new GCache();
  // code responsible of adding objects and initializing their
  // versions to 1 at the time they first read from the datastore
  Object around(PEGraph<PersistedList, Object> g, Object o):
    set(Object Object.*, o, Object) &&
    path(g, PersistedList p -/-> o)
  {
    Object ret = proceed(g, o);
    List visited = new List();
    for(PersistedList p : g.start()) {
      if(visited.notContains(o)) {
        visited.add(o);
        if(p.equal(p.version(o), gCache.version(o))) {
          p.persist(o);
          p.incVersion(o);
          gCache.incVersion(o);
          for(Object a : g.ancestors(o)) { // notify o's owners
            if(p.equal(p.version(a), gCache.version(a))) {
              p.incVersion(a);
              gCache.incVersion(a);
            } else { throw exception; }
          }
        } else { throw exception; }
      }
    }
    return ret;
  }
}
```

Figure 7.4: Cascading version-based locking policy in PePAL

Whenever a given thread attempts to change an object `o` that is persisted by the persistent list `p`, the advice compares `p.version(o)` and `GCache.version(o)`.

If these versions are equal, the advice invokes the method `p.persist(o)` and updates both versions of `o` in `p` and in `GCache`. Otherwise, this means that `o` has been changed by another thread. Then, the advice raises an exception that indicates the current version of `o` is old, which requires reapplying the change in the correct version of `o`. This procedure is also done for all owner objects of `o`. This functionality is achieved by the `CascadingVLPolicy`, which is shown in Figure 7.4.

```
class ObserverPatternAspect {
  Object around(PEGraph<Customer, Object> g, Object o):
    set(Object Object.*, o, Object) && path(g, Customer c -/-> o)
  {
    Object ret = proceed(g, o);
    for(Customer c : g.start())
      c.notiyObservers();
    return ret;
  }
}
```

Figure 7.5: Using PePAL to aspectize the observer design pattern

7.4 Observer Design Pattern

The solution of observer design pattern is almost the same as the one of the problem of field-based locking mechanism in Section 7.2. The difference is that instead of locking fields, the owner object's observers must be notified.

Consider the `ObserverPatternAspect` aspect that is shown in Figure 7.5. The resulting graph may have more than one start object, all of type `Customer`, and one end object, the changed object. Only the exact owner `Customer` objects are selected, so there is no need to collect all `Customer` objects and to check them at each advice execution to obtain the right owner as it was the case in the workaround solution in Chapter 3.

7.5 Chapter Summary

In this section, PePAL was used to provide elegant and expressive solutions to the problems mentioned in Chapter 3. The above solutions also show how PEP works as an abstraction over non-local object relationships to eliminate the need for complex object graph traversal on the developer behalf. PEP shifts this problem to the programming language level where certain optimization mechanisms can be applied.

Chapter 8

Related Work

This thesis is related to different domains of computer science. In the following sections, these related domains are discussed in some detail and compared to the here presented work.

Object persistence and how it is achieved by means of object-oriented solutions is discussed in Section 8.1. This includes the programming languages that have persistence as a language feature as well as the object-oriented persistence frameworks. In Section 8.2, the available aspect-oriented based solutions for object persistence are going to be discussed.

In Section 8.3, the thesis discusses the current research work that solves the problem of non-locality of join point properties. The need for expressive pointcut language and the research effort in providing expressive pointcuts are discussed in Section 8.4.

Section 8.5 discusses the path expression technique and its application in programming languages and different technologies. Section 8.6 discusses the related work where path expressions are applied to aspect-oriented domain to aspectize the traversal-based crosscutting concern.

Section 8.7 will compare the here presented theory and type systems to the state of the art.

8.1 Object Persistence

Providing persistence middleware service to the objects is not a trivial task for the programming languages designers. The object-oriented community has proposed a large number of solutions to object persistence. These proposals vary from explicit features of the corresponding programming languages to separated tools that could be used to add persistence to the objects of an application based on a specific language.

8.1.1 Persistence as a Language Feature

The first type of solutions is called persistence programming languages. In these techniques, the programming language has an explicit feature that could be used to provide persistence to its objects. PS-Algol [Atkinson et al. (1983a)] is considered to be the first programming language that supports orthogonal persistence. It contains additional functions that are used to transfer data between the heap and the database such as opening and closing database connections, saving objects or deleting them. These functions are added to the original language S-Algol [Cole and Morrison (1982)].

Thereafter, the programming language community invented a lot of effort to add object persistence to the object-oriented languages. For example, orthogonal persistence was added to Ada programming language [Institute (1983); Taft and Duff (1997)] by Crawley and Oudshoorn (1994); Oudshoorn and Crawley (1996). Additionally, the consequences of this extension was studied. Object persistence was introduced to the C++ by many proposals, e.g., Andlinger et al. (1991); Aritsugi and Makinouchi (2000); Evrendilek et al. (1995); Kleindienst et al. (1996); Reverbel and Maccabe (1997); Schmidt and Bauknecht (1989); Shapiro et al. (1989); Xu et al. (2000). Other examples for persistence are presented in Smalltalk [Goldberg (1984)], e.g. Copeland and Maier (1984); Hosking et al. (1993); Maier et al. (1986); Merrow and Laursen (1987). Moffat (1988) proposed a persistent Prolog language. Pascal based database language is presented in Schmidt and Matthes (1992).

The most interesting effort in this direction are the projects that aim to add orthogonal persistence to Java [Gosling et al. (1996)]. Atkinson et al. (1996) started the project called PJava, which aims to provide orthogonal persistence for the Java language without modifying the language. Roots of persistence have been defined, where individual objects can be registered during run-time. All objects reachable from a persistent root are made persistent (persistence by reachability). This is achieved by modifying the Java Virtual Machine. In addition to PJava, there are a number of other proposals to add persistence to Java, e.g. [Marquez et al. (2000); Reese (1997); SUN (2006b)].

The main problem with these solutions is that they defeat the principle of orthogonal persistence for many reasons. For instance, the developers still have to define the types that are to be persistent as well as actively calls to the persistence interface methods. This in turn requires the code of persistence to be spread all over the business logic of the applications, which is discouraged from a software engineering point of view for the understandability and maintainability of the application.

8.1.2 Object-Oriented Persistence Frameworks

There are a number of object-oriented frameworks, which are used to provide persistence services for Java objects. Some examples of these frameworks have been discussed in Chapter 2 and Chapter 3.

The problem description in Chapter 3 has illustrated that EJB [Sarang et al. (2001); SUN (2006a)] does not provide orthogonal persistence at the type level and code level. At the type level, EJB restricts the developers to follow complex rules in order to define persistent types. This breaks the type orthogonality principle since types that do not follow these rules are not eligible for persistence. Moreover, forcing developers in that way breaks the persistence independence principle, where they should know the internal structure of the framework. At the code level, the situation is the same. EJB defeats persistence independence and type orthogonality principles. In both cases, from the software engineering point of view, EJB does not support the reusability of the code.

JDO [Jordan and Russell (2003); Roos (2002)] provides a better solution for persistence than EJB. However, JDO still breaks the principle of orthogonal persistence. At the type level, the developers have to announce the persistent classes and to adjust some of them to receive persistence, e.g., implementing some `JDOHelper` methods in the persistence capable classes or including a default constructor. This will prevent the imported third-party code from being persistent, which means that the developers are obliged to prepare the imported types for persistence. This defeats the principles of independence and orthogonality.

At the code level, the developers in JDO have to designate persistence objects explicitly. Moreover, they have to know some aspects of the framework, which in turn does not attempt persistence independence. This is because JDO has a separate memory model that must be understood by the developers, and the application code must be prepared to deal with this model in order to receive persistence.

As in JDO and EJB, Hibernate [Bauer and King (2005); RHM (2007)] compromises the principles of orthogonal persistence at the type and code levels. Persistent types have to be announced explicitly and prepared for persistence. Persistence operations must be issued by the developers also. Such designation of persistent types, persistent objects, and persistence operations breaks the reusability of the code as well as the types.

8.2 Aspect-Oriented Programming for Object Persistence

The separation of the persistence concern is one of the famous examples in the aspect-oriented software development literature. There is a large effort in the aspect-oriented programming community in order to provide aspectized solutions to object persistence. Chapters 2 and 3 discussed some of them. In the following, a comparison will be presented about the current aspect-oriented programming persistence solutions.

8.2.1 A Case of AOP on Failure Recovery

The application of AspectJ in the separation of transactions was investigated for the first time by Kienzle and Guerraoui (2002), which is based on the OPTIMA framework [Kienzle (2001)]. The motivation is about to find the extent to what

AspectJ, as a prominent language example of aspect-oriented programming, can provide fully transparent transactions. The study was divided into three steps, which can be summarized as follows:

- Aspectizing transaction semantics. The authors claimed that the programmer should care about transactions in situations where it can not be concluded that transactional semantics must be applied. For example, in order to solve the problem of detecting the places where transactions should occur in a fully transparent way, the developers have to write general pointcut definitions, which cause the uniformity problem.

- Aspectizing transaction interfaces. I.e., the transaction demarcation operations `begin`, `commit`, and `rollback`. The authors showed some examples where encapsulating these interfaces in specific aspects may result in a complex and confusing code. For example, it is required to specify the transactional methods. Each method needs to be associated with an aspect instance using the `perTarget` modifier. Furthermore, according to the OPTIMA framework, each transactional object must be associated with corresponding concurrency control and recovery manager instances, which in the worst-cases can end up in poor performance.

- Aspectizing transaction mechanisms. The authors studied how to ensure the ACID properties of transactions. They concluded that AspectJ can provide elegant solutions despite that they must be implemented with care, since, e.g., they may yield in non-extensible aspects that do not absorb the changes in the base code.

The implementation of the transaction semantics using persisting containers and path expressions pointcuts overcomes the problem of the generalization. The detection of the places where the transactions are assumed to occur becomes easier since there is no need to adapt the transactional objects at the type level, e.g. to let them implement the `Serializable` interface. Instead, it is determined by the relationship between the objects and their containers where this relationship can be accessed by the path pointcuts.

Moreover, path expressions pointcuts are expressive enough to absorb the changes in the base code. The developers are able also to associate different ad-hoc functionalities such as different concurrency control policies with different persisting containers. Each persisting container applies its associated functionality to all objects that belong to this container. This in turn fulfills the OPTIMA requirements.

However, as observed by the database community, this thesis agrees with the authors' second claim. This is because the abstraction over object relationships provided by persisting containers and path expressions pointcuts does not cover the problem of specifying the transactional methods.

8.2.2 Distribution and Persistence in AOP (DPA)

The DPA framework was discussed in Section 2.5.1. Despite the importance of this work to show the applicability of aspect-oriented programming in solving object persistence, this solution does not comply with the orthogonal persistence principles.

In general, this solution does not scale well, since it is, in most parts, application-specific and its reusability is limited to applications that have a similar structure to the one of the Health Watcher system. First of all, the types that could be persistent should follow certain rules, i.e. they must implement the `Serializable` interface. Furthermore, identifying persistent objects relies on property-based crosscutting [Gybels and Brichau (2003)] that follows predefined programming conventions such as the mutator methods to start with "`set`" and persistent types to end with "`Record`". These points defeat the principle of persistence independence that promotes the reusability of the framework as well as type orthogonality. Moreover, it was shown in Chapter 3 how this framework fails to comply with the persistence by reachability principle. On the other hand, the transactional method members of certain types should be specified upfront by the programmer in separate interfaces.

However, as a solution to some of these problems, the authors of DPA have pointed out the possibility of using code analysis and generation tools that help to generate parts of aspect code that identify the transactional methods. For example, Tilevich et al. (2003) proposed an aspectization of the distribution concern by using of the XDoclet code generation engine [Walls and Richards (2003)] to generate AspectJ code that adapts the application to the distribution concern conventions.

Relying on object relationships, path expressions pointcut can provide more reusable solutions without breaking the type orthogonality. At the code level, object relationships support elegant orthogonal solutions to the persistence operations, especially retrieve, update, and delete operations. This is done by storing persistent objects in suitable persisting containers. The case of the designation of transactional method is similar to the one in Kienzle and Guerraoui (2002).

8.2.3 Persistence as an Aspect (PAA)

This work was the first attempt to provide a complete aspect-oriented persistence framework. Section 8.2.3 intrduced this framework. As pointed out in Chapter 3, it fails to fulfill the principles of orthogonal persistence in many of its facets almost exactly as it is the case in DPA.

First of all, this solution suffers from the problem of generalization, where the pointcut specifications are general, and it would not be easy to treat persistent objects differently. Moreover, the use of the persistent root principle makes it difficult to import third-party code that contains classes that must be persistent. This breaks the principle of type orthogonality.

Due to the problem of non-locality of the join point properties, this solution defeats the principle of reachability which is easy to solve by means of path expressions pointcut. The author also experienced some difficulties in providing full transparent persistence for delete and update operations.

8.2.4 Java Aspect Components (JAC)

As in the previous two solutions, JAC (cf. Section 2.5.3) fails to attempt the principles of orthogonal persistence for many reasons such as breaking type orthogonality due to the use of the persistent root concept and the persistence by reachability principle because of the problem of non-locality.

8.2.5 Other Aspect-Oriented Solutions

There are also some other proposals to aspectize middleware services related to persistence but they do not focus on aspectizing persistence itself. However, it is worth to discuss them.

As mentioned above, Tilevich et al. (2003) proposed the GOTECH framework that benefits from EJB to provide persistence service to the persistent objects after transferring them to EJB entities. The framework is developed in AspectJ, XDoclet code generation engine [Walls and Richards (2003)] and NRMI (Natural Remote Method Invocation) [Tilevich and Smaragdakis (2003)] technologies. GOTECH is evolved to the so-called JBoss-AOP [RHM (2006)] framework. Kiselev (2002) aspectized a simple database application with hard-coded SQL statements. Nagy et al. (2005) investigated the composition of aspects that share a single join point. They implement a simple persistence aspect that covers the update operations on persistent objects (whose types extend a persistent root) at such shared join points [Sicilia and Garcia-Barriocanal (2006)].

Kienzle and Gélineau (2006) presented a design overview of a possible aspect-based solution to the problem of concurrency control. The authors provided an informal discussion on the relationships and the interactions between the different aspects of the framework and showed some guidelines about the possible implementations of them and the shortcomings of the current aspect-oriented languages to fulfill the required implementation. One can consider this work as an extension to Kienzle and Guerraoui (2002).

KALA [Fabry and D'Hondt (2006)] is a domain-specific aspect language that provides aspect solutions for advanced transaction management (ATMS) [Elmagarmid (1992)] in Java applications that use the 3-tier EJB server. Charfi et al. (2006) proposed an approach to control the interactions between different aspects at the same join point. Among these aspects, the authors considered the emerging process of the notification and the persistence aspects they provided. Bodkin and Lesiecki (2005); Choi (2000) proposed an implementation of aspectizing EJB applications.

It is worth to mention that the first steps in the direction of aspect-oriented databases are introduced by Rashid (2004); Rashid and Pulvermueller (2000).

These works have an orthogonal direction to the one presented in this thesis: They are concerned with aspect persistence, while the thesis is concerned with using aspect-orientation for object persistence.

The assessment in Chapter 3 applies also to these solutions, since the types and the code need to be prepared for persistence. Moreover, these solutions suffer from another serious problem, they break the principle of reachability at the object level. This problem results from the problem of non-local join point properties that are based on object relationships, which are not supported by current aspect-oriented systems.

8.3 Non-Locality of Join Point Pproperties

Hanenberg (2005) considered the non-locality of join point properties as one of the design dimensions of aspect-oriented programming languages and systems. The author has illustrated the problem by means of an example of the observer pattern [Gamma et al. (1994)]. The author illustrated how non-local ownership information between objects complicates the modularization of the observer pattern in terms of aspects.

There are different types of non-local join point properties. A lot of research effort has been carried out to provide access to some of these types. For example, in AspectJ, which is based on Java, the call-stack information is considered as non-local. In order to gain access to call-stack information, AspectJ provides the `cflow` pointcut and its variations `pcfolw` and `cfolwbelow`. Most of aspect-oriented systems support the control flow pointcut and its variations.

There are recurrent situations where imposing aspects depends on the previous values of certain data items. An example of such situation is in the web security concern. In order to ensure the security over a request that opens a window with sanitized data, the aspect must be sure that this data comes from a secure source. Masuhara and Kawauchi (2003) proposed the `dflow` (dataflow) pointcut in order to provide the non-local values of data that occurred in the past.

Another kind of non-local join point properties is the execution trace of a program. There are many proposals to get access to this property. For example, Walker and Viggers (2004), proposed the so-called *declarative event pattern* as means to implement communication protocols that depend on the history information of the program. The authors extended AspectJ by two constructs, namely, `tracecut`, and `history`. The declarative event pattern given in the `tracecut` declaration is passed as an argument to the `history` pointcut designator that matches it with the pattern of events in the actual execution.

Other trace-based solutions have been discussed by Avgustinov et al. (2005); Douence et al. (2001, 2002, 2004b). These works argued about the importance of selecting and adapting the join points based on execution trace matching. Some works proposed the use of *stateful* aspects to define conditions based on finite state transitions to trigger advice executions on a protocol sequence of join points [Douence et al. (2004a); Vanderperren et al. (2005)].

The context-aware aspects [Tanter et al. (2006)] provide means to access information that is associated with certain contexts that are currently available or occurred in the past. Accordingly the behavior of the aspects depends on the surrounding context. The authors have motivated the need for a pointcut construct that supports addressing the available context either currently or was collected from the past at the join points. This type of non-local join point properties has also been used by Cottenier and Elrad (2005) to provide the so-called contextual pointcut expressions. These expressions generalize the semantics of `cflow` to enable advice to retrieve a richer set of context information along the call path to a target join point.

Unlike these solutions, path expressions pointcut provides access to the nonlocal information that is based on the object relationships. In fact, except Hanenberg (2005), this problem was not mentioned before. Hence, there was no solution for it other than to produce heuristic and complex workarounds by means of the reflective mechanism.

8.4 Expressiveness of the Pointcut Languages

One requirement of any aspect-oriented programming language and construct is to have expressive join point models that reflect the mental model of the developer. Stein et al. (2006) proposed the so-called *join point designation diagrams*

(JPDD) that are used to express join point selection at early stages of the software development process. The authors have followed there work by developing a tool that is used to translate these diagrams into AspectJ code [Hanenberg et al. (2007)]. The main difference to this thesis is that they consider expressiveness of the pointcut languages in the design level, whereas this thesis is concerned with the implementation level.

Moreover, Ostermann et al. (2005) argued that the expressive pointcuts increase the modularity, since they are robust to absorb any changes to the application features and compositions. The authors followed their previous remark about pointcuts that access dynamic properties of the program [Bockisch et al. (2005)] by implementing an ambitious aspect-oriented language called Alpha. The pointcuts in Alpha are Prolog queries over a database consisting of different semantic models of the program execution such as program execution trace and the heap.

Alpha maintains the whole heap in its database along with other parts of the program. In contrast to that, the path expression pointcut provides a dynamic manipulation of the needed object information at a given join point. That is, a small part of the current heap is saved for the time of the pointcut evaluation and only the matched one is exposed to the aspect.

In contrast to logic-based pointcuts of Alpha that rely on an underlying database, the path pointcut relies on traversing the heap to obtain relevant object information in the form of paths. Alpha predicates can be used to compose pointcuts that represent the notion of path pointcuts. However, these compositions may result in complex pointcut definitions that can be avoided by using the path pointcut. Moreover, one of the main goals of path pointcuts is to apply the path expressions technique in AOP as an explicit construct and to discuss the effects of this integration and how to resolve them.

There are a number of other works that discussed the importance of expressive pointcuts. For example, Rajan and Sullivan (2003) differentiated between the type level and the object level modification of the base code by the aspect. They discussed the need for aspect-oriented systems that have expressive pointcut languages and support dynamic weaving. Douence et al. (2006) argued about how current aspect systems are not expressive and efficient enough to address crosscutting concerns in the C applications at the operating system level such as network protocols and security. The authors proposed an extension to Arachne, a dynamic weaver for C applications. Masuhara and Aotani (2006) investigated how expressive pointcuts can help significantly to figure out the interactions between aspects.

The need for expressive pointcuts was also discussed by Havinga et al. (2005). The authors introduced a way to superimpose annotations to base programs using expressive pointcut constructs. Last but not least, Störzer and Hanneberg (2005) proposed an interesting classification of pointcut constructs and illustrated how expressiveness of a construct is important to find out whether a pointcut is adequate for a given situation. Consequently, this classification can be used as a guide line in the design of new pointcut constructs.

Following that aim, path pointcut is able to absorb the changes in the application code and its class structure. As discussed in Chapter 4, path expression patterns need not to be changed, if new classes are added, unless these new classes must participate in the object reference criteria of the path patterns.

8.5 Path Expressions

Path expressions technique was first introduced by Campbell and Habermann (1974) in order to synchronize the operations on data objects. In other words, they specify how threads are allowed to perform a sequence of message sends on a given object. A large number of researchers added new features to pure path expressions, such as parallel paths [Czaja (1978)] or predicate paths [Andler (1979)]. Other work has added path expressions to programming languages ranging from conventional to parallel programming languages, e.g. Pascal [Campbell and Kolstad (1979)].

Path expressions became accepted as a concise syntactical means to reference objects. This is why it became a central ingredient of object-oriented query languages such as the EJB query language (EJB-QL) [Sarang et al. (2001); SUN (2006a)] and the JDO query language [Roos (2002)]. The technique was applied by Frohn et al. (1994); Kifer et al. (1992) in querying the objects in object-oriented databases. Henrich and Robbert (2001) used path expression in a query language for structured multimedia databases.

On the other hand, other proposals have been achieved to extend and optimize the path expressions usage in the database field. For example, Van den Bussche and Vossen (1993) provided an extension to path expressions by abbreviating them depending on the knowledge of aggregation and inheritance relationships in a given schema. Ioannidis and Lashkari (1994) optimized the evaluation of path expression queries in object-oriented databases by reconstructing them to remove any ambiguities. Ozkan et al. (1995) proposed a heuristic mechanism to optimize the use of path expressions in object-oriented database queries.

The W3 Consortium introduced the XPath language [Clark and Derose (1999)] and XQuery [Boag et al. (2007)] in order to address parts of an XML document [Bray et al. (1998)]. Applying path expressions in structured documents was also discussed by Sengupta (1998, 1999) to show how a simple version of path expressions, which they proposed, increases the expressive power of query languages for structured document databases like XML and Standard Generalized Markup Language (SGML) [ISO (1986)].

This thesis studies the benefits of applying path expressions in increasing the expressiveness of pointcut languages to address object relationships at runtime and to provide aspects with access to this information. This thesis illustrates how path expressions in aspect-oriented programming enriches the field by facing some new problems and recovering them in a formal way.

8.6 Path Expressions in Aspect-Oriented Programming

Adaptive programming (AP) [Lieberherr et al. (2004); Orleans and Lieberherr (2001)] and strategic programming (SP) [Lämmel et al. (2003)] provide interesting notions similar to path expressions. They provide the developer with traversal control with the help of the so-called *traversal strategies* and *traversal schemes*, respectively.

The key concept behind adaptive programming is to provide a better separation of the object graphs *traversal-related concerns.* An example of such a crosscutting concern is when the developer wants to traverse a set of data objects in the object graph and perform specific actions when an object is visited. Without traversal strategies, the only way to do that is to add proper methods in every class in the application. This is a complex implementation that also leads to scattered and tangled code.

Traversal strategies are defined to operate on the class graphs to eliminate the potentially non-matching class paths. The result is a traversal graph that is a subset of the given class graph. According to the traversal graph, when the traverse request is issued during the execution of the program, the object graph is traversed. Whenever an object is visited by a visitor component responsible for the traversing the object graph, some visitor methods can be executed. The visitor method is either of the form `before(T t)` or `after(T t)`. This matches the semantics of `before` and `after` advice, e.g. in AspectJ, where `t`, of type `T`, is an object being visited.

The idea behind the aspect versions of adaptive programming [Lieberherr and Lorenz (2004)] and strategic programming [Kalleberg and Visser (2006)] is that the advice is executed whenever the visitor component visits an object that belongs to a path that matches the given traversal. DAJ [DAJ (2007)], pronounced “doudge”, is a tool that couples DemeterJ [Lieberherr and Orleans (1997)], DJ [Orleans and Lieberherr (2001)] and AspectJ. It defines the visitor methods as `before(ClassName)`, `after(ClassName)`, `strat()`, `finish()`, and `getReturnValue()`. These methods are invoked before visiting the object, after visiting the object, at the beginning of the traversal, at the end of traversing all fields of the root object, and after finishing the traversal to return the result, respectively. DAJ defines also a pointcut `traversal(s)` that selects all join points in the traversal defined by the strategy `s`.

This is in contrast to the path pointcut that participates in the selection of the join point and exposes the matching object paths as well. Moreover, the path pointcut extends the binding and context exposure mechanisms of aspect-oriented programming, whereas, DAJ has no means to expose object paths. This means that traversal strategies cannot be used to solve problems that this thesis addresses.

On the other hand, path pointcuts can provide elegant solutions to object traversal problems. For example, the motivating example in [Lieberherr et al. (2004)] can be solved as shown in Figure 8.1 (assuming that whenever a traversal is required, the developer invokes a special method `numWPeople()` on a `BusRoute` object).

```
void around(PEGraph g):
  execution(* BusRoute.numPeople(..))
  && path(g, BusRoute b -/-> BusStop s -/-> Person p)
{
  for(BusRoute b: g.start()) {
    write(g.end().size());
  }
  proceed(g);
}
```

Figure 8.1: Using PEP in adaptive programming

In addition to that, the thesis provides a concrete discussion to how path expressions are applied for aspect-oriented programming as an explicit pointcut construct with its informal and formal specifications.

It must be mentioned that path expressions in aspect-oriented modeling level was proposed by Stein et al. (2004). The authors provided a visualization for direct and indirect object references in aspect-orientation.

8.7 Formal Semantics and Type System

Integrating the path expressions approach into any area requires a formal description of what this integration means in order to clear any ambiguities to understand it. This is important, e.g., to simplify the implementation as well as for efficient use of path expressions. The complex nature of path expressions is the source of ambiguity.

For example, Ioannidis and Lashkari (1994) and Frohn et al. (1994) proposed an extension to object query languages by means of path expressions and provided a formal semantics for it. Also, a denotational formal semantics for XPath is proposed by Wadler (2000). Based on it, W3C published the formalization of XQuery and XPath [Draper et al. (2007)]. As a final example, a formal semantics for an extension to path expressions for XML that addresses ordering issues of the XML document elements is proposed by Murata (2001).

In general, the main difference between these works and the one presented in this thesis is that they are based on a tree data model, which contains no cycles in contrast to the graph data model. Moreover, this thesis, similarly, tries to provide a formal semantics for the meaning of the integration of path expressions and aspect-oriented programming, and to date, it is the first attempt in this direction.

Moreover, type-correctness for path queries in XML and object-oriented query languages has been discussed in many works. Some of these works have features that can be utilized in the PEP type system despite of the difference in the applied area.

For example, Colazzo et al. (2005) divided the XML trees into forests to gain control for the paths that do not participate in the result of the query and accordingly identified such occurrences as errors. Such techniques can be modified to deal with cycles in a complete type system for PEP. A type system for ownership between objects was proposed by Aldrich and Chambers (2004) to allow the programmers to specify ownership domains between objects within classes without considering the reachability. Boyapati et al. (2002) introduced a concept of lock levels to help order locks statically by means of ownership types to identify those objects which are not shared by threads (so they require no locks). Lu and Potter (2005) provided a system that describes the ownership as well as reachability in term of regions similar to ownership domains [Aldrich and Chambers (2004)]. The regions are related to each other by means of reachability rules that do not support the cyclic reachability.

As aspect-orientation becomes more popular, there is a significant effort to investigate the semantics of the aspect-oriented languages. These proposals differ from the here presented semantics as far as none of them provides a suitable base system to formalize the PEP and how it addresses the non-locality of object information.

The only previous work in aspect-orientation that presented a model for object graphs is the one for the traversal strategies by Lieberherr et al. (2004). Nevertheless, the did not consider formal semantics for the path expressions in aspect-oriented programming. They provided a notion of static analysis of the class graph similar to [Boyapati et al. (2002); Colazzo et al. (2005); Lu and Potter (2005)]. Accordingly, one can avoid traversing the non-potential matching paths.

For example, Walker et al. (2003) specified a semantics for the aspect language MinAML based on translating an aspect-oriented language into a labeled core language. The same idea was used recently by Augustinov et al. (2007) to provide a semantics for static pointcuts, which is in contrast to the dynamicity feature of the path pointcut. Wand et al. (2004) for instance, have developed a denotational semantics for pointcuts and advice in a small aspect calculus, which is used in this thesis as a target language to integrate path pointcut. MiniMAO [Clifton and Leavens (2006)], another core aspect-oriented calculus, investigates the semantics of `proceed` statement and the soundness over advice weaving. In addition to Douence and Teboul (2004), these proposals discussed the non-locality issues formally. However, these systems focused on the call-stack and trace-based non-local properties but not on object relationships that are the focus of the formal semantics presented in this thesis.

Dantas and Walker (2006) considered a calculus for harmless advice that is formalized using an information flow analysis that prevents the base program computation from being affected by the computation in advice. Clifton and Leavens (2005) also considered the semantics of the around advice and proceed. Both did not consider the pointcuts. Last but not least, Bruns et al. (2004) described μABC, a name-based calculus. They considered aspects as primitive computational entities apart from objects.

A number of researches have addressed typing issues for pointcuts and advice in order to increase the expressiveness of aspect languages type systems. Such studies became the main topic in the workshop series of FOAL [Leavens (2002)].

For example, Modular aspects with ownership (MAO) [Clifton et al. (2007)] is introduced as a model that can be used by programmers to reason about the effect of aspects on their programs. This is done by separating the objects owned by the base program from those owned by the aspects. Such a technique can be used in the evaluation of path pointcuts in the situations where objects that are part of the aspect are referenced by the persistent lists in the motivating examples of this thesis.

Ligatti et al. (2006) presented a type system for pointcuts and advice of MinAML. Despite of that their previous work [Walker et al. (2003)] applies aspects in functional programming, Ligatti et al. (2006) considered other orthogonal features that can be used as a base to investigate aspects in their extended language such as providing object typing as an orthogonal feature.

Polymorphism has been studied in the implementation of the functional language Aspectual Caml [H. Masuhara (2003)]. An intuitive simple type system that considers non-matching pointcuts was proposed by Aotani and Masuhara (2007). The pointcut type was defined in a similar way to the one presented here. Aspect FGJ (AFGJ) [Jagadeesan et al. (2006)] is an aspect-oriented calculus, which extends Featherweight GJ [Igarashi et al. (2001)] with forms for advice declaration and `proceed` whose type system supports parametric aspects. Their focus lay on the specification and correctness proof of weaving.

Chapter 9

Discussion

This chapter presents a detail discussion about the solution presented here, some design choices, and limitations.

9.1 Design Choices and Limitations

In this section, a number of design choices and points of limitations regarding the work of this thesis are being clarified.

- It must be mentioned that this thesis is not about providing a complete proposal or framework to solve object persistence. Nevertheless, it precisely discusses and addresses the lack of supporting orthogonal persistence by means of current aspect-oriented programming approaches.
- The path expression pointcut is *not only* dedicated to solve problems in the domain of object persistence, but nevertheless, this construct can be used elegantly to solve any problem in any domain where abstraction over non-local object information is necessary. For example, the observer design pattern and some traversal problems such as the ones of the traversal strategies in adaptive programming.
- The here presented approach to order multiple advice executions permits developers to specify an ordering of variable bindings using ordinary methods of the base language. Since the base language is Turing-complete, the

definitions of ordering methods do not guarantee the termination characteristic. Although this could be considered to be a weakness of the approach, the thesis still relies on this approach for two main reasons. First, defining an ordering method can be considered to be a trivial task. Hence, it could not be the case that the developer is overstrained with a definition of a terminating ordering method. Second, until now, it is difficult to determine common abstractions of what ordering schemes are typically desired by the developer (e.g., using forward or backward rules based on field information that is proposed by Lieberherr et al. (2004), or providing special constructs similar to the `declare precedence` construct that is provided by AspectJ).

- The mathematical object model that is presented in this thesis did not consider the construction of the object graphs and their dynamic transformation from a given state to another one. This is because such specification is part of the underlying programming language and not part of any formalization of the path expression pointcut.

- As the proposed formal semantics was directed to typed languages, it can be also applied to dynamic languages by simply removing type-related specifications.

- A complete formalism of the construct may include a specification of the advice execution ordering mechanism. However, such requirements might be considered as fine-tuning since the current semantics is enough to solve the complex problems that are used to motivate this work.

- The presented formalization also did not considered static enhancements such as relying on the class graph statically to predict the potential matching paths from the object graph dynamically. Such mechanism is used in [Lieberherr et al. (2004)]. Despite of the importance of such enhancement with respect to performance issues, it is also considered as a further work that is not needed to prove the feasibility of the solution.

- It is important to say that the intention behind providing the type system for `path` pointcuts is to prove their static safety. However, a complete type system must provide the dynamic behavior of the construct such as the advice matching and the evaluation rules for the new language terms like the ones that belong to the `PEGraph` interface. Proving the preservation property will be a must.

Chapter 10

Discussion and Conclusion

This chapter presents in Section 10.1 an overall discussion about the proposed solution in this thesis. Then, it presents a discussion on future work in Section 10.2. Section 10.3 discusses the contributions of this thesis. Finally, Section 10.4 summaries the work of the thesis in a chapter-wise manner.

10.1 Discussion

As the previous chapters have presented the problem of non-local object information in aspect-oriented programming, and proposed a solution for this problem by means of path expression pointcuts, it is essential to discuss some important issues about the impact of the solution in solving the problem and its design choices and limitations.

First of all, it is needed to remind that since PEP addresses the problem of non-local object information, it is considered as a binder pointcut like `target` and `this` pointcuts. Therefore, its functionality is not to select join points, rather, to transfer information from the join point context to the aspect context.

Chapter 4 motivated the use of path expressions to address the non-local object information in aspect-oriented programming. However, it is possible to apply path expressions in a way other than extending the pointcut language with PEP. This can be achieved by providing a generic API or tool that takes a path expression as an argument and searches the object graph for the matching paths. However, the provision of this access by using an explicit language construct, i.e.

PEP, has a great advantage over the use of the generic API. The reason is that the explicit language constructs are inner parts of the programming languages, e.g. the Java virtual machine. Accordingly, it is possible to apply available optimization techniques to the language constructs in the level of the programming language, which in turn provides better performance.

There are good reasons to address the question of PEP performance. First, PEP is applied on the object graph, which is known as a complex data structure. The evaluation of PEP may require a large part of the object graph to be parsed in order to find the matching paths. Despite of the existence of many graph traversal algorithms, there are still some problems that affect the performance of PEP.

One such problems is how to determine dynamically the start point in the object graph for the traversal. The general case is to start from the root of the object graph, normally, the main object. In this case, the algorithm may need to traverse through a large amount of objects until it reaches the target object that is specified by the path pattern, and consequently finds a matching path.

However, in many cases it is possible to get the traversal start object from the aspect itself depending on the required functionality of the aspect. This minimizes the overhead and limits the traverse context to a restricted part of the object graph instead of the whole one. For example, to start the traversal from the list of the persisting containers that are used in this thesis for solving the object persistence operations (cf. Chapter 7).

Another reason for a potential performance leak is that object graphs in most cases contain cycles. In this case the evaluation of PEP in order to find the matching paths to the given path expression pattern may result in an infinite number of matching paths. This process may not terminate. For this reason, this thesis assumes some restrictions in order to ensure the termination. It uses cycle detection algorithms to restrict the detected cycles from being parsed again except when the required variable bindings are not fulfilled yet. The evaluation algorithm depends on the last variable in the path pattern to decide whether the required variable bindings are already evaluated. However, the resulting `PEGraph` will contain the cycle in the matching paths, hence, the infinite number of paths are already exposed to the advice. This means that the developers will be able to access this piece of information.

As a further solution for the performance problem, one can apply static analysis to the class graph and build some static assumptions of the potential matching paths to the given path pattern in the pointcut. From this analysis, another graph is to be established, which is going to guide the object graph traversal dynamically. Such mechanism is used in adaptive programming [Lieberherr et al. (2004)], where this graph is known as traversal graph. This means that each path expression pattern will have its own traversal graph. Despite of the importance of such enhancement with respect to performance issues, it is also considered as a further work that is not needed to prove the feasibility of the solution, which was proved by the proposed PEP implementation.

The size of the exposed `PEGraph` objects may still be relatively big. One may argue that the advice implementer will be obliged to face the problem of traversing this part of the object graph again. However, the exposed `PEGraph` has an easy-to-use generic interface that provides the developer with proper traversal forward and backward methods, the access to the relation fields, and the access to the list of inner objects. Furthermore, the structure of the `PEGraph` is determined by the criteria given by the developer in the path expression pattern. So, the developer is aware of the object information that is needed to be accessed.

For example, if the developer specifies the following path pattern: `A a -/-> B b`, she/he can easily accessed the two related objects `a` and `b` by using the `start` and `end` members of the resulting `ExPEG<A, B>`, respectively. Moreover, the use of the "`/`" wildcard reflects the fact that there is no need to access special relationship information neither there is a need to access the correct types of the inner objects in the matching paths. The list of inner objects provides the required access to all objects that occur in the matching path between `a` and `b`.

Another issue regarding the solution is the presented approach to order multiple advice executions in Section 4.2.2.5. It permits the developers to specify an ordering of variable bindings using ordinary methods of the base language. Since the base language is Turing-complete, the definitions of ordering methods do not guarantee the termination characteristic. Although this could be considered to be a weakness of the approach, the thesis still relies on this approach for two main reasons. First, defining an ordering method can be considered to be a trivial task. Hence, it could not be the case that the developer is over strained with a definition of a terminating ordering method. Second, until now, it is difficult to determine common abstractions of what ordering schemes are typically desired by the developer (e.g., using forward or backward rules based on field information that is proposed by Lieberherr et al. (2004), or providing special constructs similar to the `declare precedence` construct that is provided by AspectJ).

A number of notes are related to the formal semantics of PEP that is presented in this thesis. First, the notations given here are chosen to simplify the formalization, e.g., meaningful names where used to denote the semantic functions instead of symbols. Another issue is that the thesis did not provide a formal proof for this semantics. However, a number of examples have been used to show its applicability. In addition to that, this formal semantics has been integrated with a complete semantics of an existing pointcut language, and a number of examples have been used to show how this integration works.

The proposed formal semantics was directed to typed languages, nevertheless, it can be also applied to dynamic languages by simply removing type-related specifications.

The formal specification of PEP did not consider the construction of the object graphs and their dynamic transformation from a given state to another one. This is because such specification is part of the underlying programming language and not a part of any formalization of the path expression pointcut. PEP assumes, as it is ever the case, that it gets the access to the object graph when needed from the execution unit of the target programming language.

It is important to say that the intention behind providing the type system for path pointcuts is to show how correct types for PEP ingredients are evaluated statically. However, a complete type system must provide the dynamic behavior of the construct such as the advice matching and the evaluation rules for the new language terms like the ones that belong to the `PEGraph` interface. Proving the progress and the preservation properties will be a must.

Finally, there are a number of points that need to be mentioned. First, this thesis is not about providing a complete proposal or framework to solve object persistence. Nevertheless, it precisely discusses and addresses the lack of supporting orthogonal persistence by means of current aspect-oriented programming approaches. Moreover, PEP provides access to the objects and their relationships. Since this access is so important for object persistence, PEP can be used heavily in any aspect-oriented solution for object persistence.

Another important point is that PEP is *not only* dedicated to solve problems in the domain of object persistence, but nevertheless, this construct can be used to provide elegant solutions for any problem in any domain where abstraction over non-local object information is necessary. For example, the observer design pattern, the visitor design pattern, the problem of object queries in the heap, and some traversal problems such as the traversal strategies in adaptive programming.

In this thesis, the integration of PEP into the existing pointcut languages was restricted to the aspect-oriented programming languages that are based on counterpart object-oriented languages, e.g. the AspectJ language. This raises a question about the applicability of PEP to other pointcut languages that are based on programming languages from other paradigms. For example, one may think about how can PEP be applied to the logic-based pointcut languages like CARMA [Gybels and Brichau (2003)] and ALPHA [Ostermann et al. (2005)].

In ALPHA, path pointcuts can be achieved by writing composed pointcuts of multiple `reachable/2` predicate or `reachablevia/3` pointcut designator, which determines the reachability through a given object. However, such composed pointcuts tend to be complex and syntactically large, which in turn making them difficult to maintain and in most cases non-reusable. In addition to that, it would be difficult to determine the association names between the related objects by using these predicates. A better solution is to add a new pointcut with which one can specify path expressions. Direct relationship between two objects can be added to the ALPHA database. A possible syntax for such pointcut might be `path/5(Type1, source object, relation, Type2, target object)`. Then, a query can use these relationships to address the relationships between more than two objects as it is the case with PEP.

10.2 Future Work

Path expression pointcut has enriched the field of aspect-oriented programming, since it consists of a number of interesting, yet complex concepts. Last chapter discussed some limitations and design issues regarding the work that is presented here. In the following, a number of suggestions to further work are discussed.

Object persistence. A very promising direction for future work is to utilize the path expression pointcut along with the notion of ad-hoc persisting containers to provide a complete aspect-oriented solution for object persistence that supports the orthogonal persistence principle.

Ownership and object query in object-orientation. Path expression pointcut can be applied to solve problems that deal with the ownership relation, reachability, and object sharing in object-oriented programming such as [Pearce and Noble (2006); Rayside et al. (2006); Willis et al. (2007)]. Further work in aspectizing such solutions is feasible, since the path expressions technique is considered as a good abstraction for the object relationships.

Other crosscutting concerns. Another direction is to investigate the impact of the path expression pointcut in aspect-oriented programming in order to provide better solutions for different problem domains that require abstraction over non-local object information. For example, in aspectizing the *visitor design pattern* [Gamma et al. (1994)].

Formal semantics. This thesis provides a denotational formal semantics that describes the meaning of the PEP and its result. For a complete understanding of the concept, an operational formal semantics would be necessary to specify how the PEP is evaluated.

Static analysis. The current specification of the PEP introduces a number of assumptions and restrictions to minimize the complexity of parsing the object graphs and ensuring the evaluation termination. However, some further work is required to find the impact of the static information in minimizing the complexity. One promising idea is to use the notion of class graphs that is used in [Lieberherr et al. (2004)] to perform static matching instead of doing the whole matching dynamically.

Type system. One direction of further research is to consider completing the type system for PEP. For example, to add dynamic typing rules for the manipulation of object graphs, the decision upon reachability, the advice lookup mechanism, and the term evaluation of the `PEGraph` ingredients. Such work will enrich the field, since it will introduce a number of interesting typing issues to the aspect-oriented programming. Such type system will be first to discuss typing issues for object graphs and the access to the non-local objects in the typed aspect-oriented systems.

PEP integration. The here presented implementation is done by providing a minimal aspect-oriented language that is dedicated to prove the feasibility of the PEP implementation. However, it would be easy to use the PEP formal semantics and type system that are presented in this thesis to implement the PEP in other languages, e.g. AspectJ, AspectC++, and AspectS.

10.3 Summary of Contributions

This thesis investigated and addressed the lack of obliviousness support in current aspect-oriented persistence systems at the program and the language level. A similar situation is discussed also with respect to the observer design pattern, which is considered as a typical crosscutting concern. The thesis' claim was that the rationale behind this problem is that current pointcut languages do not support the non-local object information at the join points. As a solution, this thesis proposed a new pointcut construct called path expression pointcut, which provides expressive means to abstract over the non-local objects and object relationships.

In the aspect-oriented literature, object persistence is considered as a typical example of a crosscutting concern that can be addressed by aspect-oriented programming techniques. An obvious hypothesis behind this claim is that the obliviousness property of aspect-oriented programming [Filman and Friedman

(2000)] meets the orthogonal principle of object persistence [Atkinson and Morrison (1995)]. Following this hypothesis, a large number of proposals have introduced aspect-oriented solutions for object persistence. As a consequence, it is important to find the extent to which these proposals provide better solutions for object persistence that fulfill the principle of orthogonal persistence. This leads to the first contribution:

> **Contribution 1:** Assessment of how existing aspect-oriented persistence solutions [Pawlak et al. (2004); Rashid and Chitchyan (2003); Soares et al. (2002)] fulfill the orthogonal persistence principle. For the purpose of understandability and comparability, this assessment also includes some current prominent object-oriented persistence approaches [Bauer and King (2005); Jordan and Russell (2003); SUN (2006a)].

The thesis concluded that current aspect-oriented persistence approaches compromise the principle of orthogonal persistence. The main reason is that object persistence depends heavily on object relationships, however, current aspect-oriented programming languages do not support the quantification of persistence related join points that is based on non-local object information. The result was that the quantification over non-local object information is an important issue that needs to be addressed by current pointcut languages. This is exactly in line with many other contributions that dealt with different types of non-local join point properties. Moreover, to improve the obliviousness level in any aspect-based persistence solution, this thesis proposed the use of persistence as an ad-hoc functionality by means of the so-called persisting containers.

As a solution, the thesis proposed the use of the well-known path expressions technique. This is provided as an explicit pointcut construct called path expression pointcut. Aspect-oriented programming includes a number of sophisticated concepts and mechanisms such as binding, context exposure, weaving, and advice execution. Therefore, the integration of path expressions and aspect-oriented programming raises a number of interesting issues. This is about the second contribution:

> **Contribution 2:** Detail discussion of the effects of the application of the path expressions technique to aspect-oriented programming. This includes proper extensions to the concept of parameter bindings and their unification property, the mechanism of context exposure, and the advice execution mechanism. In addition to that, new semantics have been introduced to aspect-oriented programming like the path matching process and the resulting parts of the object graphs. In order to show the feasibility of those concepts, a prototype implementation for the `path` pointcut is proposed.

A number of examples have been given to illustrate all facets of the new construct and the new semantics of some aspect-oriented programming concepts. This includes also addressing how path expression patterns are being matched against the object graphs, how these selected paths are exposed to the aspect's pointcuts and advice in terms of the so-called path expression graphs (`PEGraph`) objects, and how concrete pointcuts that are making use of the path pointcut are evaluated. Such informal description for this collection of complex structures and concepts is not sufficient to provide a precise and unambiguous understanding of the concept. This in turn leads to the next contribution:

> **Contribution 3:** Providing a formal semantics for the `path` pointcut and how it is possible to integrate this formalization with the formal semantics of a complete pointcut language.

A denotational formal semantics was provided to clear any ambiguity to understand the path expression pointcut and how this construct addresses the non-local object information at a given join point. It was built over a mathematical model of the object graph in terms of the theory of sets. This formalization has shown the meaning of the `PEGraph` construct, how it is constructed, and the way it is bound and provided to the context exposure mechanism. Also, this formalization has shown how this resulting binding is unified with the bindings resulting from other pointcut designators like the `target` pointcut designator. This formal semantics can be used to guide further development of the `path` pointcuts. In addition to that, it can work as a guideline to formalize similar concepts.

In general, this thesis is convinced that in order to provide a new pointcut language construct it is necessary to introduce an unambiguous semantics for it. This formalization in turn becomes beneficial for other researchers. Consequently, the here presented formal semantics for PEP is considered as a step into the right direction.

Exposing `PEGraph` objects to the advice in order to make the later accesses the relevant part of the object graph to the selected join point requires providing a suitable interface for the `PEGraph`. Since most of the current aspect-oriented programming languages are based on typed languages, a proper type for `PEGraph` must be provided in order to benefit from the type information that is already provided by the path expression pattern. Such type is addressed by the following contribution:

> **Contribution 4:** Type-correctness for the `path` pointcut is provided by means of a simple type system that ensures the safety of producing the `PEGraph` type.

This thesis provides a type system for the `path` pointcut and proves the progression property of the type system with regard to the static evaluation of a proper `PEGraph` type. The static typing rules have been used to show how the correctness of a given `PEGraph` object along with its ingredients is proved by means of derivation.

Using PePAL language, the thesis showed how path expression pointcuts can provide elegant and robust solutions for the problems of current aspect-based persistence frameworks. The concurrency control policies that are given by means of the `path` pointcuts can be integrated easily into the existing persistence frameworks. For example, before and after the `trapUpdates` pointcut in PAA framework [Rashid and Chitchyan (2003)], the proper concurrency control aspect from the here presented solutions can be triggered. Such triggers can be applied in the same way as presented by Kienzle and Guerraoui (2002) for `preOperation` and `pastOperation` pointcuts.

Moreover, `path` pointcut along with persisting containers can be used efficiently to solve the problem of uniformity [Al-Mansari et al. (2007a); Kienzle and

Gélineau (2006)]. This can be achieved by two steps: First, attaching an individual middleware service to each persisting container, e.g. define a persisting container that provides pessimistic field-based locking policy to all of its contained objects and dedicate another persisting container to provide optimistic version-based locking policy to its contained objects. Second, by means of the path expression pointcut, one can perform the right policy for each object depending on the reachability between the persisting containers and the persistent object being manipulated.

In this direction, the thesis has shown how such mechanism can help with respect to the retrieve, update, and delete persistence operations. With respect to the create operation, the created persistent object must be put inside a suitable persisting container so that beyond this point the three other persistence operations can be achieved obliviously. Identifying persistent objects at the instantiation time cannot be achieved in fully oblivious way, however, e.g., one can use an interactive tool that can be run in the testing phase before the application delivery. The user of such tool should decide which object instantiation must be persisted.

10.4 Conclusion Summary

In the following, the whole work of the thesis in the previous chapters is summarized.

Chapter 2 Background. This chapter presented an introduction to the field of programming for separation of concerns. Then it introduced the field of aspect-oriented software development while concentrating on aspect-oriented programming as the specific field of the motivating problem. In this thesis, all code examples were given in terms of AspectJ, so it was necessary to introduce this language. As a background chapter, the problem domain, i.e. object persistence, was introduced while covering most important issues related to it as well as the main features of desirable object persistence solution that meets the obliviousness property of aspect-oriented programming.

Chapter 3 Problem description. This chapter discusses the problems that prevent current aspect-oriented systems from providing persistence solutions that comply with the principle of orthogonal persistence. The assessment in this chapter showed that the main reason for this lack is that these pointcut languages do not support accessing the non-local object information, which is considered to play the main role in any solution to object persistence. Moreover, these systems suffer from the so-called problem of generalization. The chapter also showed how object relationships are important when aspectizing other crosscutting concerns. The problem statement has been provided precisely at the end of this chapter.

Chapter 4 Path expression pointcuts. This chapter addresses the problem of getting access to the non-local object information at the join points by proposing path expression pointcut as an explicit pointcut construct that extends the current pointcut languages. It illustrated the extension by means of plenty of examples that described the different facets of the construct and other issues that were introduced by applying path expressions to aspect-oriented programming. For example, the extension to the parameter binding and context exposure mechanisms or the advice execution mechanism.

Chapter 5: Formal Semantics and Type System. This chapter proposed an unambiguous denotational semantics for path expression pointcuts. This formalization was introduced on top of a precise mathematical model for the object graphs in term of the theory of sets. This semantics was integrated into complete pointcut language semantics to provide a clear meaning for PEP in a whole language. Then, this chapter proposed proper types for the PEP and its ingredients like the path expression graph and the parameter binding list, and a simple type system for PEP is provided thereafter.

Chapter 6: Implementation. This chapter presented a prototype implementation of PEP in a typed object-oriented language called PePAL. The discussion covered different facets of the implementation and provided various examples that illustrated the usage of the construct in this language.

Chapter 7: Motivating Examples Revisited. Using PEP in PePAL, this chapter provided solutions to the problems mentioned in Chapter 3.

Chapter 8: Related Work. This chapter discussed the related work to this thesis. The related work was divided into several domains: conventional solutions for object persistence, aspect-oriented solutions for object persistence, non-locality issues in aspect-oriented programming, expressiveness of pointcut languages, path expressions and their applications, path expressions in aspect-oriented programming, formal semantics and type systems for aspect-oriented programming languages in particular, and those related to the formalisation of the path expressions and the object graphs.

Chapter 10: Discussion and Conclusion. This chapter discussed the proposed solution of this thesis. Then, it concluded this thesis with a summary of future work, a discussion of the main contributions of the thesis, and an overview of the work done in each chapter of this thesis.

Bibliography

ACPP. Aspectc++ homepage, 2007. URL `http://www.aspectc.org/`.

A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986. ISBN 0-201-10088-6.

M. Al-Mansari and S. Hanenberg. Path Expression Pointcuts: Abstracting over Non-Local Object Relationships in Aspect-Oriented Languages. In *NODe '06: Proceedings of the Net.ObjectDays conference*, volume 88 of *Lecture Notes in Informatics*, pages 81–96, Erfurt, Germany, September 2006. Gesellschaft fr Informatik.

M. Al-Mansari, S. Hanenberg, and R. Unland. Orthogonal Persistence and AOP: a Balancing Act. In *ACP4IS '07: In Proceedings of 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software at AOSD 2007*, Vancouver, Canada, March 2007a. ACM Press. Article nr. 2.

M. Al-Mansari, S. Hanenberg, and R. Unland. Aspect-Oriented Programming: Selecting and Exposing Object Paths. In *SC '07: Proceedings of the 6th international symposium on software composition*, Braga, Portugal, March 2007b. Springer-Verlag. ISBN 978-3-540-77350-4.

M. Al-Mansari, S. Hanenberg, and R. Unland. On to formal semantics for path expression pointcuts. In *SAC '08: Proceedings of the 23rd ACM Symposium on Applied Computing*, Fortaleza, Ceará, Brazil, March 2008. ACM Press, to appear.

J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP '04: Proceedings of the 18th European Conference Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Oslo, Norway, June 2004. Springer-Verlag.

S. Andler. *Predicate Path Expressions: a High-Level Synchronization Mechanism.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1979.

P. Andlinger, C. Gierlinger, and G. Quirchmayr. Making C++ Object Persistent by Using a Standard Relational Database System. In *DEXA '91: Proceedings of the International Conference in Database and Expert Systems Applications*, pages 374–379, Berlin, Germany, 1991. Springer-Verlag.

T. Aotani and H. Masuhara. Towards a type system for detecting never-matching pointcut compositions. In *FOAL '07: In Workshop on Foundations of Aspect-Oriented Languages FAOL*, pages 23–26, Vancouver, British Columbia, Canada, March 2007. ACM Press. ISBN 1-59593-671-4.

M. Aritsugi and A. Makinouchi. Multiple-type Objects in an Enhanced C++ Persistent Programming Language. *Software, Practice and Experience*, 30(2): 151–174, 2000.

ASPJ. Aspectj programmers guide, 2007. URL `http://eclipse.org/aspectj/`.

M. P. Atkinson. Persistence and Java - a Balancing Act. In *Proceedings of International Symposium on Objects and Databases*, pages 1–31, Sophia Antipolis, France, June 2000. Springer-Verlag.

M. P. Atkinson. Persistent Foundations for Scalable Multi-Paradigmal Systems. In *IWDOM '91: International Workshop on Distributed Object Management*, pages 26–50, Edmonton, Alberta, Canada, August 1992. Morgan Kaufmann.

M. P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.

M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, 1983a.

M. P. Atkinson, K. Chisholm, W. P. Cockshott, and R. Marshall. Algorithms for a Persistent Heap. *Software: Practice and Experience*, 13(3):259–271, 1983b.

M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An Orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, December 1996. ISSN 0163-5808.

P. Augustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of Static Pointcuts in AspectJ. In *POPL '07: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 11–23, Nice, France, January 2007. ACM Press.

P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an Extensible AspectJ ompiler. In *AOSD 05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, Chicago, Illinois, USA, March 2005. ACM Press.

C. Bauer and G. King. *Hibernate in Action.* Manning Publications Co., New York, 2005. ISBN 1932394-15-X.

L. Benavides, M. Śudholt, W. Vanderperren, B. Fraine, and D. Suvée. Explicitly Distributed AOP Using AWED. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 51–62, Bonn, Germany, March 2006. ACM Press.

S. Boag, D. Chamberlin, M. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: an XML Query Language.* W3C Recommendation, January 2007. URL `http://www.w3.org/TR/xquery/`.

C. Bockisch, M. Mezini, and K. Ostermann. Quantifying over Dynamic Properties of Program Execution. In *DAW '05: In 2nd Dynamic Aspects Workshop*, page 7175. ACM Press, March 2005.

R. Bodkin and N. Lesiecki. Enterprise Aspect-Oriented Programming with AspectJ, March 2005. Tutorial at AOSD05.

C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA '02: Proceedings of the annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 211–230, Seattle, Washington, USA, November 2002. ACM Press.

T. Bray, J. Paoli, C. M. Sperberg-McQueen., E. Maler, and F. Yergeau. *Extensible Markup Language.* W3C Recommendation, 1998. URL `http://www.w3.org/TR/REC-XML`.

G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μabc: a Minimal Aspect Calculus. In *CONCUR '04: 15th International Conference Concurrency Theory*, pages 209–224, London, UK, August-September 2004. ACM Press.

P. Butterworth, A. Otis, and J. Stein. The Gemstone Object Database Management System. *Communications of the ACM*, 34(10):64–77, 1991.

R. Campbell and A. Habermann. The Specification of Process Synchronization by Path Expressions. In *Symposium on Operating Systems*, volume 16, pages 89–102. Springer-Verlag, 1974.

R. H. Campbell and R. B. Kolstad. Path Expressions in Pascal. In *ICSE '79: Proceedings of the 4th International Conference on Software Engineering*, pages 212–219, Munich, Germany, September 1979. IEEE Computer Society.

L. Cardelli. Type Systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, Boca Raton, FL, USA, 1997. ISBN 0-8493-2909-4.

R. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standards ODMG 3.0.* Morgan Kaufmann Publishers, New York, 2000. ISBN 1-55860-647-5.

M. Ceccato and P. Tonella. Adding distribution to existing applications by means of aspect oriented programming. In *SCAM '04: Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 107–116, Chicago, IL, USA, 2004. IEEE Computer Society.

A. Charfi, M. Riveill, M. Blay-Fornarino, and A. Pinna-Dery. Transparent and dynamic aspect composition. In *SPLAT Workshop at AOSD '06*, pages 26–50, Bonn, Germany, March 2006.

S. Chiba and K. Nakagawa. Josh: an open aspectj-like language. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 102–111, Lancaster, England, March 2004. ACM Press.

R. Chitchyan, A. Rashid, and P. Sawyer. Comparing requirements engineering approaches for handling crosscutting concerns. In *In Workshop on Requirements Engineering (held with CAiSE)*, Porto, Portugal, August 2005.

Jung Pil Choi. Aspect-oriented programming with enterprise javabeans. In *EDOC '00: Proceedings of the 4th International Enterprise Distributed Object Computing Conference*, pages 252–261, Makuhari, Japan, September 2000. IEEE Computer Society.

J. Clark and S. Derose. *XML Path Language (XPath).* W3C Recommendation, 1999. URL `http://www.w3.org/TR/Xpath`.

C. Clifton and G. Leavens. Minimao: Investigating the semantics of proceed. In *FOAL '05: . In Workshop on Foundations of Aspect-Oriented Languages FAOL*, pages 26–50, Edmonton, Alberta, Canada, August 2005. Morgan Kaufmann.

C. Clifton and G. T. Leavens. Minimao: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, 2006.

C. Clifton, G. T. Leavens, and J. Noble. Mao: Ownership and effects for more effective reasoning about aspects. In *ECOOP '07: Proceedings of European Conference on Object-Oriented Programming*, volume 4609 of *LNCS*, pages 451–475, Berlin, Germany, July-August 2007. Springer-Verlag.

G. Clossman, P. Shaw, M. Hapner, J. Klein, R. Pledereder, and B. Becker. Java and relational databases: Sqlj (tutorial). In *SIGMOD '98: Proceedings ACM*

SIGMOD International Conference on Management of Data, page 500, Seattle, Washington, USA, June 1998. ACM Press.

T. Cohen and J. Gil. Aspectj2ee = aop + j2ee. In *ECOOP '04: Proceedings of the 18th European Conference Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 219–243, Oslo, Norway, June 2004. Springer-Verlag.

D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for path correctness of xml queries. In *SEBD '05: Proceedings of the Thirteenth Italian Symposium on Advanced Database Systems*, pages 264–271, Brixen-Bressanone, Italy, June 2005. ISBN 88-548-0122-4.

A. Cole and R. Morrison. *An Introduction to Programming with S-Algol*. Cambridge University Press, 1982. ISBN 0-52125-001-3.

G. Copeland and D. Maier. Making smalltalk a database system. In *SIGMOD '84: Proceedings of the ACM/SIGMOD International Conference on the Management of Data*, pages 316–325, Boston, Massachusetts, June 1984. ACM Press.

T. Cottenier and T. Elrad. Contextual pointcut expressions for dynamic service customization. In *DAW '05: In 2nd Dynamic Aspects Workshop*, page 7175. ACM Press, March 2005.

P. J. Courtois. On time and space decomposition of complex structures. *Communications of the ACM (CACM)*, 28(6):590–603, 1985.

S. Crawley and M. Oudshoorn. Orthogonal persistence and ada. In *TRI-Ada '94: Proceeding of the 1994 Conference on TRI-Ada*, pages 298–308, November 1994.

L. Czaja. Implementation approach to parallel systems. *Information Processing Letters*, 7(6):291–295, 1978.

DAJ. Daj home page, 2007. URL `http://daj.sf.net/`.

D. Dantas and D. Walker. Harmless advice. In *POPL '06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 383–396, Charleston, South Carolina, USA, 2006. ACM Press.

J. Darlington, A. Field, and H. Pull. The unification of functional and logic languages. In *Logic Programming: Functions, Relations, and Equations*, pages 37–70. Prentice-Hall, 1986. ISBN 0-13-539958-0.

E. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, New Jersey, 1976.

R. Douence and L. Teboul. A pointcut language for control-flow. In *GPCE '04: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 95–114, Vancouver, Canada, October 2004. Springer-Verlag.

R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Reflection '01: Proceedings of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Kyoto, Japan, September 2001. Springer-Verlag.

R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE '02: Proceedings of Generative Programming and Component Engineering Conference*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Pittsburgh, PA, USA, October 2002. Springer-VErlag.

R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 141–150, Lancaster, UK, March 2004a. ACM Press.

R. Douence, P. Fradet, and M. Ś'udholt. *Trace-based aspects*, page 201217. Addison-Wesley, 2004b.

R. Douence, T. Fritz, N. Loriant, J. Menaud, M. Ségura-Devillechaise, and M. S´udholt. An expressive aspect language for system applications with arachne. *Transaction on Aspect-Oriented Software Development I*, 3880:174–213, 2006.

D. Draper, P. Fankhauser, M. Fernndez, A. Malhotra, K. Rose, M. Rys, J. Simon, and P. Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics*. W3C Recommendation, January 2007. URL `http://www.w3.org/TR/xquery-semantics/`.

A. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992. ISBN 1-55860-214-3.

R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, third edition, 2000. ISBN 0-201-54263-3.

C. Evrendilek, A. Dogac, and T. Gesli. A preprocessor approach to persistent c++. In *ADBIS '95: roceedings of the Second International Workshop on Advances in Databases and Information Systems*, Workshops in Computing, pages 235–251, Moscow, June 1995. Springer.

J. Fabry. *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, July 2005.

J. Fabry and T. D'Hondt. Kala: Kernel aspect language for advanced transactions. In *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1615–1620, Dijon, France, April 2006. ACM Press. ISBN 1-59593-108-2.

R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2005. ISBN 0-32121-976-7.

R. E. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA Workshop on Advanced Sep. of Concerns*, Minneapolis, MN, Oct, October 2000.

J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *VLDB '94: Proceedings of 20th International Conference on Very Large Data Bases*, pages 273–284, Santiago de Chile, Chile, September 1994. Morgan Kaufmann. ISBN 1-55860-153-8.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0-20163-361-2.

A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison Wesley, 1984. ISBN 0-20111-372-4.

J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996. ISBN 0-201-63451-1.

J. Gray. The Transaction Concept: Virtues and Limitations. In *VLDB '81: Proceedings of the International Conference on Very Large Data Bases*, pages 144–154, Cannes, France, September 1981. IEEE Computer Society. ISBN 0-934613-15-X.

J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks in a large shared data base. In *VLDB '75: Proceedings of the International Conference on Very Large Data Bases*, pages 428–451, Framingham, Massachusetts, USA, September 1975. ACM Press.

J. Gray, T. Bapty, S. Neema, D. C. Schmidt, A. S. Gokhale, and B. Natarajan. An approach for supporting aspect-oriented domain modeling. In *GPCE '03: International Conference on Generative Programming and Component Engineering*, volume 2830 of *LNCS*, pages 151–168, Erfurt, Germany, September 2003. Springer-Verlag.

K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 60–69, Boston, MA, USA, 2003. ACM Press.

A. Yonezawa H. Masuhara, H. Tatsuzawa. Aspectual caml: an aspect-oriented functional language. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, page 320330, Tallinn, Estonia, September 2003. ACM. ISBN 1-59593-064-7.

S. Hanenberg. *Design Dimensions of Aspect-Oriented Systems.* PhD thesis, Duisburg-Essen University, Essen, Germany, October 2005.

S. Hanenberg and R. Unland. Parametric introductions. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 17 – 21, Boston, MA, USA, March 2003. ACM Press.

S. Hanenberg, R. Hirschfeld, and R. Unland. Morphing aspects: Incompletely woven aspects and continuous weaving. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 46–55, Lancaster, UK, March 2004. ACM Press.

S. Hanenberg, D. Stein, and R. Unland. Roles from an aspect-oriented perspective. In *VAR'05: Views, Aspects and Roles Workshop, ECOOP 2005*, Glasgow, UK, July 2005.

S. Hanenberg, D. Stein, and R. Unland. From aspect-oriented design to aspect-oriented programs: tool-supported translation of jpdds into code. In *AOSD '07: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 49–62, Vancouver, British Columbia, Canada, March 2007. ACM. ISBN 1-59593-615-7.

B. Harbulot and J. R. Gurd. A join point for loops in aspectj. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 63–74, Bonn, Germany, March 2006. ACM Press.

W. Havinga, I. Nagy, and L. Bergmans. Introduction and derivation of annotations in aop: Applying expressive pointcut language to introductions. In *In European Interactive Workshop on Aspects in Software*, 2005.

A. Henrich and G. Robbert. Poqlmm: A query language for structured multimedia documents. In *MDDE '01: Proceedings 1st International Workshop on Multimedia Data and Document Engineering*, Lyon, France, July 2001.

E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, Lancaster, UK, March 2004. ACM Press.

R. Hirschfeld. Aspects - aspect-oriented programming with squeak. In *NetObjectDays '02: Proceedings of International Conference NetObjectDays - Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232, Erfurt, Germany, October 2003. Springer-Verlag. ISBN 3-540-00737-7.

A. Hosking, E. Brown, and J. B. Moss. Update logging for persistent programming languages: A comparative performance evaluation. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 429–440, Dublin, Ireland, August 1993. Morgan Kaufmann. ISBN 1-55860-152-X.

A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *TOPLAS*, 23(3):396–450, 2001.

American National Standards Institute. *The Programming Language Ada Reference Manual*. Springer, 1983. ISBN 3-540-12328-8. ANSI/MIL-STD-1815A-1983.

Y. E. Ioannidis and Y. Lashkari. Incomplete path expressions and their disambiguation. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 138–149, Minneapolis, Minnesota, USA, May 1994. ACM Press.

ISO. Standard generalized markup language (sgml), 1986. International Organization for Standardization, ISO 8879: Information Processing - Text and Office Systems.

A. Jackson, P. Sánchez, L. Fuentes, and S. Clarke. Towards traceability between ao architecture and ao design. In *EA '06: In Workshop of Early Aspects at AOSD 2006*, Bonn, Germany, March 2006.

R. Jagadeesan, A. Jeffrey, and J. Riely. Typed Parametric Polymorphism for Aspects. *Science of Computer Programming*, 63(3):267–296, 2006.

D. Jordan and C. Russell. *Java Data Objects*. OReilly Media, first edition, 2003. ISBN 0-59600-276-9.

K. Kalleberg and E. Visser. Combining aspect-oriented and strategic programming. *Electr. Notes Theor. Comput. Sci.*, 147(1):5–30, 2006.

A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Prentice Hall, 1994. ISBN 0-13-629239-9.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwing. Aspect-oriented programming. In *ECOOP '97: Proceedings of European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyv´askyl´a, Finland, June 1997. Springer-Verlag.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with aspectj. *Communications of the ACM (CACM)*, 44(10): 59–65, 2001a.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP '01: In Proceedings of European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science (LNCS)*, pages 327–353, Budapest, Hungary, June 2001b. Springer-Verlag.

J. Kienzle. *Open Multi-threaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming.* PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 2001.

J. Kienzle and S. Gélineau. Ao challenge - implementing the acid properties for transactional objects. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 202–213, Bonn, Germany, March 2006. ACM Press.

J. Kienzle and R. Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *ECOOP '02: Proceedings of European Conference on Object-Oriented Programming*, volume 2374 of *LNCS*, pages 37–61, Malaga, Spain, June 2002. Springer-Verlag.

M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *SIGMOD '92: Proceedings ACM SIGMOD International Conference on Management of Data*, pages 393–402, San Diego, California, USA, June 1992. ACM Press.

H. Kim and S. Park. Two version concurrency control algorithm with query locking for decision support. In *ER '98 Workshops: Workshops on Data Warehousing and Data Mining, Mobile Data Access, and Collaborative Work Support and Spatio-Temporal Data Management*, volume 1552 of *Lecture Notes in Computer Science*, pages 157–168, Singapore, November 1998. Springer.

E. King. IBM report on the contents of a sample of programs surveyed, 1978. San Jose, CA, USA, IBM.

I. Kiselev. *Aspect-Oriented Programming with AspectJ.* SAMS, Indianapolis, IN, USA, 2002. ISBN 0-67232-410-5.

J. Kleindienst, F. Plasil, and P. Tuma. Lessons learned from implementing the corba persistent object service. In *OOPSLA '96: Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 150–167, San Jose, California, USA, October 1996. ACM Press.

I. Krechetov, B. Tekinerdogan, A. Garcia, C. Chavez, and U. Kulesza. Towards an integrated aspect-oriented modeling approach for software architecture design. In *In 8th International Workshop on Aspect-Oriented Modeling, AOSD '06*, Bonn, Germany, March 2006.

R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Mannning Publications, 2003. ISBN 1-93011-093-6.

C. Lamb, G. Landis, J. A. Orenstein, and D. Weinreb. The objectstore database system. *Communications of the ACM (CACM)*, 34(10):50–63, 1991.

R. Lämmel, E. Visser, and J. Visser. Strategic programming meets adaptive programming. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 168–177, Boston, MA, USA, March 2003. ACM Press.

G. Leavens. Foal 2002 workshop, April 2002. URL `http://www.eecs.ucf.edu/~leavens/FOAL/index-2002.html`.

K. Lieberherr and D. Lorenz. *Coupling Aspect-Oriented and Adaptive Programming*, page 145164. Addison-Wesley, 2004.

K. Lieberherr and D. Orleans. Preventive program maintenance in demeter/java. In *ICSE '97: Proceedings of the International Conference on Software Engineering*, page 604605, Boston, Massachusetts, USA, May 1997. ACM-Press.

K. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(2):370–412, 2004.

J. Ligatti, D. Walker, and S. Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 63(3):240–266, 2006.

W. Lin and J. Nolte. Basic timestamp, multiple version timestamp, and two-phase locking. In *VLDB '83: Proceedings of the International Conference on Very Large Data Bases*, pages 109–119, Florence, Italy, October-November 1983. Morgan Kaufmann. ISBN 0-934613-15-X.

Y. Lu and J. Potter. A type system for reachability and acyclicity. In *ECOOP '05: Proceedings of European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 479–503, Glasgow, UK, July 2005. Springer-Verlag.

D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented dbms. In *OOPSLA '86: Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482, Portland, Oregon, USA, November 1986. ACM Press.

A. Marquez, J. Zigman, and S. Blackburn. Fast portable orthogonally persistent java. *Software - Practice and Experience (SPE)*, 30(4):449–479, 2000.

H. Masuhara and T. Aotani. Issues on observing aspect effects from expressive pointcuts. In *ADI '06: In Proceedings of Workshop on Aspects, Dependencies and Interactions at ECOOP '06*, pages 53–61, July 2006.

H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programmingg. In *APLAS '03: In 1st Asian Symposium on Programming Languages and Systems*, volume 2895 of *LNCS*, pages 105–121, Beijing, China, November 2003. Springer-Verlag.

K. Mens, C. Lopes, B. Tekinerdogan, and G. Kiczales. Aspect-oriented programming workshop report. In *ECOOP Workshops*, volume 1357 of *LNCS*, pages 483–496, Jyväskylä, Finland, June 1997. Springer-Verlag.

T. Merrow and J. Laursen. A pragmatic system for shared persistent objects. In *OOPSLA '87: Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 103–110, Orlando, Florida, USA, October 1987. ACM Press.

D. S. Moffat. Modular commitment in persistent prolog. In *Prolog and Databases*, pages 267–. Ellis Horwood Ltd., Chichester, U.K., 1988. ISBN 0-7458-0371-7.

C. Mohan and D. Haderle. Algorithms for flexible space management in transaction systems supporting fine-granularity locking. In *EDBT '94: International Conference on Extending Database Technology*, volume 779 of *LNCS*, pages 131–144, Cambridge, UK, 1994. Springer-Verlag.

C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *SIGMOD*

'92: Proceedings ACM SIGMOD International Conference on Management of Data, pages 124–133, San Diego, California, USA, June 1992. ACM Press.

M. Murata. Extended path expressions for xml. In *PODS '01: Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 126–137, Santa Barbara, California, USA, May 2001. ACM. ISBN 1-58113-361-8.

I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In *NODe '05: Proceedings of the Net.ObjectDays conference*, volume 69 of *Lecture Notes in Informatics (LNI)*, pages 19–38, Erfurt, Germany, September 2005. Gesellschaft fr Informatik.

OBJ. Objectivity technical overview, 2006. URL `http://www.objectivity.com/Misc/docs/oodb_techOverview.pdf`. Objectivity Inc.

D. Orleans and K. Lieberherr. Dj: Dynamic adaptive programming in java. In *Reflection '01: Proceedings of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 73–80, Kyoto, Japan, September 2001. Springer-Verlag.

K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP '05: Proceedings of European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*, page 214240, Glasgow, UK, July 2005. Springer-Verlag.

M. J. Oudshoorn and S. C. Crawley. Beyond ada 95: The addition of persistence and its consequences. In *Ada-Europe '95: International Conference on Reliable Software Technologies*, volume 1088 of *LNCS*, pages 342–356, Montreux, Switzerland, June 1996. Springer-Verlag.

C. Ozkan, A. Dogac, and C. Evrendilek. A heuristic approach for optimization of path expressions. In *DEXA '95: International Conference on Database and Expert Systems Applications*, volume 978 of *LNCS*, pages 522–534, London, UK, September 1995. Springer-Verlag.

E. Panagos, A. Biliris, H. Jagadish, and R. Rastogi. Fine-granularity locking and client-based logging for distributed architectures. In *EDBT '96: International Conference on Extending Database Technology*, volume 1057 of *LNCS*, pages 388–402, Avignon, France, March 1996. Springer-Verlag.

D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM (CACM)*, 15(8):1053–1058, 1972.

R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. Jac: an aspect-based distributed dynamic framework. *Software - Practice and Experience (SPE)*, 34(12):1119–1148, 2004.

D. J. Pearce and J. Noble. Relationship aspects. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 75–86, Bonn, Germany, March 2006. ACM Press.

C. Pierce. *Types and Programming Languages.* The MIT Press, Cambridge, Massachusetts, 2002. ISBN 0-262-16209-1.

A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 141–147, Enschede, The Netherlands, April 2002. ACM Press.

A. Popovici, G. Alonso, and T. Gross. Spontaneous container services. In *ECOOP '03: Proceedings of European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 29–53, Darmstadt, Germany, July 2003. Springer-Verlag.

H. Rajan and K. Sullivan. Need for instance level aspects with rich pointcut language. In *SPLAT Workshop at AOSD '03*, Boston, MA, USA, March 2003.

B. R. Rao. *Object-Oriented Databases, Technology, Applications and Products.* McGraw Hill, 1994. ISBN 0-07-051279-5.

A. Rashid. On to aspect persistence. In *GCSE '00: International Symposium on Generative and Component-Based Software Engineering*, volume 2177 of *Lecture Notes in Computer Science*, pages 26–36, Erfurt, Germany, October 2000. Springer.

A. Rashid. *Aspect-Oriented Database Systems*. Springer, 2004. ISBN 978-3-540-00948-1.

A. Rashid and R. Chitchyan. Persistence as an aspect. In *AOSD '03: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 120–129, Boston, MA, USA, March 2003. ACM Press.

A. Rashid and E. Pulvermueller. From object-oriented to aspect-oriented databases. In *DEXA '00: International Conference on Database and Expert Systems Applications*, volume 1873 of *LNCS*, pages 125–134, London, UK, September 2000. Springer-Verlag.

D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *WODA '06: Fourth International Workshop on Dynamic Analysis associated with the International Conference on Software Engineering (ICSE'06)*, Shanghai, China, May 2006.

G. Reese. *Database Programming with JDBC and Java*. O'Reilly, 1997. ISBN 1-56592-270-0.

F. Reverbel and A. Maccabe. Making corba objects persistent: the object database adapter approach. In *COOTS '97: Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, pages 55–76, Portland, Oregon, USA, June 1997.

RHM. *Hibernate: Relational Persistence for Java*. Red Hat Middleware, LLC., 2007. URL `http://www.hibernate.org/`.

RHM. Jboss aop, 2006. URL `http://labs.jboss.com/portal/jbossaop/`. Red Hat Middleware, LLC.

D. Ries and M. Stonebraker. Effects of locking granularity in a database management system. *ACM Transactions on Database Systems (TODS)*, 2(3):233–246, 1977.

R. M. Roos. *Java Data Objects*. Addison-Wesley, 2002. ISBN 0-32112-380-8.

K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Advice weaving in aspectj. In *AOSD ’04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 16–25, Lancaster, UK, March 2004. ACM Press.

P. Sarang, K. Gabhart, A. Tost, T. McAllister, R. Adatia, M. Juric, T. Osborne, F. Arni, J. Lott, V. Nagarajan, C. Berry, D. O’Connor, J. Griffin, A. Mulder, and D. Young. *Professional EJB*. Wrox Press, 2001. ISBN 1-861005-08-3.

D. Schmidt and K. Bauknecht. Db++ - persistent objects for c++. In *BTW ’89: Datenbanksysteme in B´uro, Technik und Wissenschaft*, pages 177–281, Z´urich, Switzerland, March 1989. Springer-Verlag.

J. W. Schmidt and E. Matthes. The database programming language dbpl rationale and report. Technical report, Universit´at Hamburg, Hamburg, Germany, Germany, 1992. ESPRIT BRA Project 3070 FIDE Technical Report FIDE/92/46.

A. Sengupta. Toward the union of databases and document management: The design of docbase. In *COMAD ’98: Proceedings of Conference on Management of Data*, Hyderabad, India, December 1998.

A. Sengupta. The complete closure: toward a unified view of structured document database objects. In *ISAS ’99: Fifth International Conference on Information Systems Analysis and Synthesis*, volume 5, pages 269–273, Orlando, Florida, USA, July-August 1999.

M. Shapiro, P. Gautron, and L. Mosseri. Persistence and migration for c++ objects. In *ECOOP ’89: Proceedings of European Conference on Object-Oriented Programming*, pages 191–204, Nottingham, UK, July 1989. Cambridge University Press. ISBN 0-521-38232-7.

M. Sicilia and E. Garcia-Barriocanal. Extending object database interfaces with fuzziness through aspect-oriented design. *SIGMOD Record*, 35(2):4–9, 2006.

S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with aspectj. In *OOPSLA '02: Proceedings of the annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 174–190, Seattle, Washington, USA, November 2002. ACM Press.

O. Spinczyk, A. Gal, , and W. Schr´oder-Preikschat. Aspectc++: An aspect-oriented extension to c++. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Sydney, Australia, February 2002. Australian Computer Society, Inc. ISBN 0-909925-88-7.

D. Stein, S. Hanenberg, and R. Unland. A uml-based aspect-oriented design notation for aspectj. In *AOSD '02: Proceedings of the International Conference on Aspect-Oriented Software Development*, pages 106 – 112, Enschede, The Netherlands, April 2002. ACM Press.

D. Stein, S. Hanenberg, and R. Unland. Query models. In *UML '04: Proceedings of the 7th International Conference on the Unified Modeling Language*, volume 3273 of *LNCS*, pages 98–112, Lisbon, Portugal, 2004. Springer-Verlag.

D. Stein, S. Hanenberg, and R. Unland. Expressing different conceptual models of join point selections in aspect-oriented design. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 15–26, Bonn, Germany, March 2006. ACM Press.

L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, second edition, 1994. ISBN 0-26219-338-8.

M. Störzer and S. Hanneberg. A classification of pointcut language constructs. In *SPLAT Workshop at AOSD '05*, Chicago, Illinois, USA, March 2005.

B. Stroustrup. *C++ Programming Language.* Addison-Wesley, third edition, 1997. ISBN 0-201-32755-4.

SUN. Enterprise javabeans technology, 2006a. URL `http://java.sun.com/products/ejb/index.jsp`. Sun Microsystems Inc.

SUN. Java^{TM} 2 platform, enterprise edition (j2ee^{TM}) specification, November 2003. URL `http://java.sun.com/products/j2ee/docs.html`. Sun Microsystems Inc.

SUN. *Java^{TM} Remote Method Invocation Specification.* Sun Microsystems, Inc., December 1999. Revision 1.7, JavaTM 2 SDK, Standard Edition, v1.3.0.

SUN. *Java Object Serialization Specification.* Sun Microsystems, Inc., 2006b. URL `http://java.sun.com./javase/6/docs/technotes/guides/serialization/index.html`.

J. Suzuki and Y. Yamamoto. Extending uml with aspects: Aspect support in the design phase. In *ECOOP Workshops '99: Proceedings of the 3rd AOP Workshop held in conjunction with ECOOP 99*, volume 1743 of *Lecture Notes in Computer Science*, pages 299–300, Lisbon, Portugal, June 1999. Springer. ISBN 3-540-66954-X.

T. Taft and R. Duff. *Ada 95 Reference Manual, Language and Standard Libraries.* Springer, 1997. ISBN 3-540-63144-5. International Standard ISO/IEC 8652.

E. Tanter, K. Gybels, M. Denker, and A. Bergel. Context-aware aspects. In *SC '06: Proceedings of the 5th international symposium on software composition*, volume 4089 of *LNCS*, pages 227–242, Vienna, Austria, March 2006. Springer-Verlag.

E. Tilevich and Y. Smaragdakis. Nrmi: Natural and efficient middleware. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 252–, Providence, RI, USA, May 2003. IEEE Computer Society. ISBN 0-7695-1920-2.

E. Tilevich, S. Urbanski, Y. Smaragdakis, and M. Fleury. Aspectizing server-side distribution. In *ASE '03: Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 130–141, Montreal, Canada, October 2003. IEEE Computer Society. ISBN 0-7695-2035-9.

J. Van den Bussche and G. Vossen. An extension of path expressions to simplify navigation in object-oriented queries. In *DOOD '93: In the 3rd International Conference on Deductive and Object-Oriented Databases*, volume 760 of *LNCS*, pages 267–282, Phoenix, Arizona, USA, December 1993. Springer-Verlag.

W. Vanderperren, D. Suve, M. A. Cibrn, and B. De Fraine. Stateful aspects in jasco. In *SC '05: Proceedings of the 5th international symposium on software composition*, volume 3628 of *LNCS*, pages 167–181, Edinburgh, Scotland, April 2005. Springer-Verlag.

M. Veit and S. Herrmann. Model-view-controller and object teams: a perfect match of paradigms. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 140–149, Boston, MA, USA, 2003. ACM Press.

A. Wadler. A formal semantics of patterns in xslt and xpath. *Source Markup Languages: Theory and Practice*, 2(2):183–202, 2000. ISSN 1099-6621.

D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 127–139, Uppsala, Sweden, August 2003. ACM. ISBN 1-58113-756-7.

R. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *FSE '12: Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 159–169, Newport Beach, CA, USA, October-November 2004. ACM. ISBN 1-58113-855-5.

C. Walls and N. Richards. *XDoclet in Action*. Manning Publications, 2003. ISBN 1-93239-405-2.

M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 26(5):890910, 2004.

V. I. Wietrzyk and M. A. Orgun. Versant architecture: Supporting high - performance object databases. In *IDEAS '98: Proceedings of the International*

Database Engineering and Applications Symposium, pages 141–149, Cardiff, Wales, UK, July 1998. IEEE Computer Society. ISBN 0-8186-8307-4.

D. Willis, D. J. Pearce, and J. Noble. Efficient object querying for java. In *ECOOP '06: Proceedings of European Conference on Object-Oriented Programming*, volume 4067 of *LNCS*, pages 28–49, Nantes, France, July 2007. Springer-Verlag.

L. Wood. Document object model (dom), version 1.0, 1998. URL `http://www.w3.org/TR/REC-DOM-Level-1/`. W3C Recommendation.

D. Xu, X. Han, J. Wang, and Y. Chen. A strategy for persistent object service under corba and web environment. In *TOOLS '00: 36th International Conference on Technology of Object-Oriented Languages and Systems*, pages 100–107, Xi'an, China, October-November 2000. IEEE Computer Society. ISBN 0-7695-0875-8.