Join Point Designation Diagrams: A Visual Design Notation for Join Point Selections in Aspect-Oriented Software Development

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

durch die Fakultät für Wirtschaftswissenschaften der
Universität Duisburg-Essen
Campus Essen

vorgelegt von
Dominik Stein
geboren in Bonn am Rhein

begutachtet durch
Prof. Dr. Rainer Unland (Universität Duisburg-Essen)
Prof. Dr. Jörg Kienzle (McGill University, Montreal, Kanada)

November 2010
Tag der mündlichen Prüfung: 24. Januar 2011

Meiner Familie: Harriet, Linus und Philip

Abstract

Sharing knowledge about program specifications is a crucial task in collaborative software development. Developers need to be able to properly assess the objectives of the program specification in order to adequately deploy or evolve a piece of program. The specification of join point selections (also known as "pointcuts") in Aspect-Oriented Software Development (AOSD) is a piece of a program which frequently tends to grow quite complex, in particular if the join point selections involve selection constraints on the dynamic execution history of the program. In that case, readers of the pointcut specification frequently find themselves confronted with considerable comprehension problems because they need to inspect and realize an intricate and fragmented program specification in order to reconstruct the true objectives of the join point selection.

This thesis presents Join Point Designation Diagrams (JPDDs) as a possible solution to the problem. JPDDs are a visual notation that provides an extensive set of join point selection means which are consolidated from a variety of contemporary aspect-oriented programming languages. JPDDs are capable of highlighting different join point selection constraints depending on the conceptual view on program execution which underlies the join point selection. With the help of these means, JPDDs are capable of representing complex join point selections on the dynamic execution of a program in a succinct and concise manner. JPDDs may be used by software developers of different aspect-oriented programming languages to represent their join point selections for the sake of an improved comprehensibility of the join point selections and – thus – for the sake of an easier communication between software developers.

This thesis gives empirical evidence that JPDDs indeed facilitate the comprehensibility of join point selections. To do so, it conducts a controlled experiment which compares JPDDs to equivalent pointcut implementations in an aspect-oriented programming language. The experiment shows that JPDDs have a clear benefit over their codified counterparts in most of the case, while only in few cases no such benefit could be measured.

Acknowledgements

This thesis would have been unachievable for me if I hadn't received generous support of others, who deserve grateful acknowledgement at this place:

Foremost, I am very grateful for the support of my wife Harriet, who took care of the kids and managed the housework while I was working on this thesis night after night, month after month, for almost more than two years. By doing that, she granted me the joys of family life while releasing me from all of its duties, and no complaint passed her lips. Harriet, I thank you so much for this generous and persisting support!

Then, I would like to give my thanks to my colleague Dr. Stefan Hanenberg. Stefan's "eternal quest" to find inconsistencies, inappropriate conclusions, or the one-point-you-haven't-thought-about in one's work is a great avail. There is *always* a point in what he's saying. And so, every discussion with him always leads to a substantial improvement of the quality and the soundness of one's work. Having him as a colleague and disputant was – and still is – a great pleasure and an invaluable privilege!

Furthermore, I would like to thank my first supervisor Prof. Dr. Rainer Unland for offering me all necessary financial and material support that I needed to pursue and complete this thesis. I truly appreciated the ultimate freedom that he granted me in conducting my research and in directing my work.

Likewise, I would like to thank Prof. Dr. Jörg Kienzle for agreeing to be my second supervisor and for travelling all the way from Montreal (Canada) to Essen (Germany) for the sole purpose of my oral examination. Prof. Kienzle gave me valuable comments which further improved this thesis.

Finally, I would like to thank my students, in particular Jens Bartelheimer and Nadezhda Avramova, for their substantial help on implementing the prototypical tools that accompany this thesis. It was a great pleasure to work with them and to see the verve with which they have completed their studies and with which they have advanced the implementation of the Modeler for Join Point Designation Diagrams (M4JPDDs). Further thanks goes to the 35 student who volunteered to participate in the empirical experiment.

Contents

Tables

Listings

Figures

Abbreviations

abc	Aspect Bench Compiler
AJDT	AspectJ Development Tools
A-LTS	Aspectual Labeled Transition System
ANOVA	Analysis Of Variance
ANTLR	Another Tool for Language Recognition
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
AOSDw/UC	Aspect-Oriented Software Development Approach with Use Cases
AOSF	Aspect-Oriented Statechart Framework
API	Application Programming Interface
approx.	approximately
bMSC	Basic Message Sequence Charts
BPMN	Business Process Modeling Notation
cf.	Latin: confer ("compare")
DEP	Declarative Event Patterns
df	degrees of freedom
DFD	Data Flow Diagram
DSL	Domain-Specific Languages
DT	Development Tools
e.g.	Latin: exempli gratiā ("for example")
EAOP	Event-Based AOP
EMF	Eclipse Modeling Framework
et al.	Latin: et alii/et aliae ("and others")
GEF	Graphical Editing Framework
GoF	Gang Of Four (E. Gamma, R. Helm, R.E. Johnson, J. Vlissides)
HiLA	High-Level Aspects
HMSC	Highlevel Message Sequence Chart
HTML	HyperText Markup Language

i.e.	Latin: id est ("that is")
IDE	Integrated Development Environment
JAsCoDT	JAsCo Development Tools
JPDD	Join Point Designation Diagrams
JVM	Java Virtual Machine
l. bound	lower bound
LTS	Labeled Transition System
M4JPDD	Modeler for Join Point Designation Diagrams
marg. mean	marginal mean
MATA	Modeling-of-Aspects-using-a-Transformation-Approach
ms	millisecond
MSC	Message Sequence Charts
MVC	Model-View-Controller
OCL	Object Constraint Language
p.	page
PEP	Path Expression Pointcuts
pp.	pages
QBE	Query By Example
RAM	Reusable Aspect Models
sd	standard deviation
SDL	Specification and Description Language
SQL	Standard Query Language
std. error	standard error
STL	Standard Template Library (C++)
TM	Tracematch
u. bound	upper bound
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1
Introduction

This chapter provides the reader with a general idea about the subject of this thesis. To do so, it shortly sketches the context, focus, contribution, and the underlying assumptions of this thesis. The chapter concludes with a detailed outline of the thesis.

1.1 Context

The research question addressed by this thesis belongs to the domain of software engineering. Software engineering is a subdomain of computer science which deals with "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" and the study of these approaches (cf. [*IEEE (1990)*]). Research in software engineering has brought up many approaches which aim to support the software developers' task throughout the phases of the software development lifecycle (e.g. in requirements elicitation and analysis, design, programming, testing, maintenance, etc.). A common goal of these approaches is to make software development and/or maintenance "easier", e.g. "easier" to specify (for example, with the help of suitable language constructs that allow for a succinct and concise specification of software), "easier" to manage (for example, with the help of potent modularization mechanisms that permit to effectively break down complexity), or "easier" to comprehend (for example, by using concepts and abstractions that are taken from the "real" world rather than from the technical solution domain).

1.2 Focus

The particular focus of this thesis is on Aspect-Oriented Software Development (AOSD) [*Filman et al. (2005)*]. AOSD introduces a new abstraction to software development, called **aspect**, which non-invasively augments an existing software specification with extra features. AOSD is particularly concerned about the specification of deeply pervading features which take effect at multiple places in a software specification. Again, the goal of AOSD is to ease the specification of the software (since developers need to define such features only once rather than at all places where they take effect), its management (since developers find the complete specification of a feature in one place, and they do not need to gather it from all over the software specification), as well as its comprehension (since developers may think of features as "aspects" of a software specification, and they may concentrate on different "aspects" one at a time).

AOSD introduces another abstraction, i.e. a **join point selection** (also known as "pointcut"), which is used to designate all places in the software specification where a pervading feature is supposed to take effect. This thesis is particularly concerned with the comprehensibility (of the specification) of such join point selections. To be more precise, this thesis is concerned with (the comprehensibility of the specification of) join point

selections which constrain the dynamic execution history of a running program. This thesis refers to these join point selections as join point selections which select *dynamic* and *behavioral* join points (for more explanations and helpful illustrations, see next Chapter 2).

1.3 Contributions

This thesis recognizes that there is a the need for a communication means which helps to comprehend the specification of join point selections that select dynamic and behavioral join points. The thesis illustrates with help of seven examples which are implemented with seven contemporary aspect-oriented programming languages why the specification of join point selections constraining the earlier execution history of a running program are difficult to understand (see Chapter 3).

In response to that, the thesis presents a new notation, called Join Point Designation Diagrams (JPDDs), which is capable to overcome the identified comprehension problems. By doing so, this thesis contributes to the body of knowledge in software engineering in three major ways:

I. The thesis recognizes that different aspect-oriented programming languages provide different selection means for the selection of dynamic and behavioral join points, yet there is only a small overlap (see section 4.1). In response to that, JPDDs provide a consolidated and consistent set of abstractions that permit to express prevalent join point selection means of many contemporary aspect-oriented programming languages in a comprehensive and uniform way.

II. The thesis recognizes that join point selections in aspect-oriented software development usually rely on particular conceptual views on program execution (see section 5.7). The thesis recognizes that different notational means are required to suit each of these views in order to facilitate the comprehension of the join point selections. In response to that, JPDDs provide different notational means to cope with three kinds of views, i.e. a state view, a control flow view, and a data flow view.

III. The thesis gives empirical evidence that JPDDs are easier to comprehend than an existing aspect-programming language with respect to the comprehension of (some facets of) data flow constraints (see Chapter 7).

Apart from these major research contributions, this thesis yields a set of minor technological and methodological contributions to the domain of software engineering:

i. The thesis is accompanied with a prototypical tool which permits to create, save, and modify JPDDs (cf. [*Bartelheimer (2006)*, *Avramova (2008)*]). The tool permits to draw JPDDs relating to any of the previously mentioned conceptual views on program execution.

ii. The thesis is accompanied with a prototypical code generator which translates JPDDs relating to a control flow view on program execution into AspectJ code, and vice versa [*Stein & Hanenberg (2008)*].

The prototypical tools are briefly introduced in Appendix B.

1.4 Assumptions

JPDDs adopt both symbols and semantics from the Unified Modeling Language (UML) [*Booch et al. (1998)*]. This decision has been made because it is assumed that most software developers are familiar with the UML or one of its predecessors, such as Data Flow Diagrams (DFD) [*DeMarco (1979)*], State Charts [*Harel (1987)*], or Message Sequence Charts (MSC) [*ITU (1999)*]. Moreover, it is assumed that software developers are accustomed to the different conceptual views on program execution which underlie these notations.

Apart from that, it is assumed that there is an exigency to specify aspect-oriented join point selections which constrain more than just a few system events in the dynamic execution history of a running program. In other words, it is assumed that there exist numerous situations where software developers need to read and understand and communicate complex join point selections which select *dynamic* and *behavioral* join points, and which require them to scrutinize and disentangle a non-trivial set of interdependencies between those relevant system events.

1.5 Thesis Outline

Chapter 2 Background: An Overview To Aspect-Orientation

This chapter first introduces the general concepts and mechanisms of aspect-oriented software development, such as **aspects**, **join points**, and **weaving**, and explains how they relate to each other. It addresses and elucidates selected key characteristics of the aspect-oriented software development approach, with a particular emphasis on characteristics which are essential to the *selection* of join points (as this is the special focus of this thesis).

Subsequently, the chapter exemplifies the concrete manifestation of the aspect-oriented concepts and mechanisms in the aspect-oriented programming language AspectJ [*Kiczales et al. (2001)*] (which is probably the most widely known aspect-oriented programming language with the largest user base). The special focus of that exemplification is on the selection of *dynamic* and *behavioral* join points, as this is the special focus of this thesis.

The chapter introduces a collection of pointcut designators for the selection of dynamic and behavioral join points and briefly sketches how these pointcut designators are enforced by the aspect-oriented weaver. The intention of doing so is that the reader acquires a firm understanding of the nature of a dynamic and behavioral join point, as this is essential for the remainder of the thesis. The chapter is by no means a thorough introduction to AspectJ in general (which can be found in [*Laddad (2003)*], for example).

Finally, the chapter recapitulates the essential facts about aspect-oriented software development, in particular about the selection of dynamic and behavioral join points, which readers have to keep in mind while reading this thesis.

Chapter 3 Problem Statement

This chapter illustrates the problem that this thesis is going to tackle. To do so, it presents four examples of join point selections of dynamic and behavioral join points and demonstrates how they can be realized with four different existing aspect-oriented programming languages. The examples are taken from existing literature on aspect-oriented

software development, and their realizations with existing aspect-oriented programming languages are non-trivial. The chapter elucidates the details of these realizations such that the reader is able to recognize the rationale of the implementations. Afterwards, the chapter discusses the problems that developers are faced with when they need to read and understand such complex realizations. The chapter concludes with a summary and a generalization of the problems.

The examples presented in this chapter are complemented with further examples, which are presented in Appendix A. The appendix presents three further examples implemented with three further aspect-oriented programming languages. These examples do not add any new arguments to the problem description. However, they give further evidence about the problem, and they are used in the discussion chapter of this thesis to evaluate the proposed solution.

Chapter 4 State Of The Art

This chapter gives an overview to the state of the art in representing join point selections (of dynamic and behavioral join points). To do so, the chapter first describes existing aspect-oriented *programming* approaches which address (some parts of) the problem outlined in the previous chapter, and discusses why these approaches are unsuited to solve the problem (in general). Illustrations of (some of) the approaches have been given in the previous chapter and in Appendix A. Thus, the description of the aspect-oriented programming approaches is kept rather short and refrains from showing any further detailed examples.

Subsequently, the chapter introduces existing aspect-oriented *modeling* approaches which (directly or indirectly) address the problem outlined in the previous chapter, and discusses why these approaches are also unsuited to solve the aforementioned problem. Unlike the previous case, this chapter introduces the aspect-oriented modeling approaches in closer detail and explains with help of illustrative examples why the approaches fail to solve the problem in a satisfactory manner. The goal is to give the reader a first impression about the existing approaches and their visual notations, as well as to give the reader a firm understanding of the undissolved deficits of these approaches.

Chapter 5 Join Point Designation Diagrams

This chapter introduces Join Point Designation Diagrams (JPDDs) as a solution to the identified problem. JPDDs are a visual means to represent join point selections, which are particularly capable of representing join point selections that select *dynamic* and *behavioral* join points. The chapter introduces all of the notational means of JPDDs which may be used to specify such join point selections. The chapter describes what properties of a system can be constrained by JPDDs, and how these constraints may be extended (e.g. using deviation means such as wildcards, regular expressions, and path expressions) in order to select a broader set of join points.

The particular focus of the chapter is on the representation of different conceptual views on program execution with JPDDs. The chapter illustrates the necessity for providing different notational means to express such conceptual views appropriately. Furthermore, it introduces the notational means that are offered by JPDDs in order to express those different views, and highlights the differences between them.

Finally, the chapter elucidates the capabilities of JPDDs to specify join point selection constraints on static and structural join points. These means are introduced for the sake of completeness only and are not taken into account in the remainder of the thesis.

The chapter concludes with the explanation of three examples which illustrate the deployment of JPDDs to express join point selections. Each of these examples reflects on a different conceptual view.

Chapter 6 Discussion

This chapter discusses why JPDDs are considered to improve the comprehensibility of join point selections (as opposed to existing approaches). The chapter expresses the motivating join point selections presented in the problem statement (see Chapter 3) using JPDDs, and compares these JPDDs to their codified counterparts. The chapter discusses why JPDDs are expected to facilitate the comprehension of the join point selections (in comparison to their codified counterparts).

In addition to the examples from the problem statement, the chapter furthermore investigates the examples presented in Appendix A. Unlike the motivating examples (presented in Chapter 3), which either relate to a data flow view or a state view on program execution, the join point selections presented in Appendix A (mostly) relate to a control flow view on program execution. The chapter discusses why the JPDD representation of these additional sample join point selections is considered to be easier to understand than their codified counterparts.

The chapter concludes with a summary of the findings of the comparisons, and points out the special suitability of JPDDs as a common communication means for software developers which permits them to express and understand the interdependencies between join point selection criteria of complex join point selections more easily and in a programming language-independent way. The summary contrasts these benefits to the costs of learning the notational means of JPDDs.

Chapter 7 Evaluation

This chapter gives empirical evidence that the conjecture made in the previous chapter, which claims that JPDDs are capable of facilitating the comprehension of complex join point selections, actually holds. To do so, the chapter reports on a controlled experiment which compares JPDDs to their textual counterparts, which has been conducted with several participants who were asked to perform different comprehension tasks on a number of join point selections. The chapter explains the details of the experiment design and presents a detailed analysis of the experiment results. The results show that JPDDs indeed facilitate the comprehension of complex join point selections in most cases. The chapter closes with an explanation of the potential threats to the validity of the experiment as well as with a summary of the conclusions that can be drawn.

Chapter 8 Conclusion

This chapter concludes the thesis. It summarizes the previous chapters, recapitulates the achievements made and points out their limitations, and gives an outlook to interesting future work.

Appendix I More Motivating Examples

This appendix complements the problem statement by presenting three further examples implemented with three further aspect-oriented programming languages. These examples do not add new arguments to the problem description. However, they give further evidence about the problem, and they are used in the discussion chapter of this thesis to evaluate the proposed solution.

Appendix II Prototypical Tool Support

This appendix reports on prototypical tool support that has been provided for the deployment of JPDDs in the software development process. The appendix presents a UML profile which may be used to draw JPDDs with most standard UML modeling tools, a modeling tool for JPDDs (realized as a plug-in for the Eclipse IDE), which features proper visualization of the JPDD-specific symbols, and a code generator, which permits to translate JPDDs into aspect-oriented program code.

Appendix III More Facts and Figures

This appendix complements the empirical evaluation of JPDDs by presenting further facts and figures. Apart from providing supplementary statistics, the appendix visualizes the basic data which has been gathered from the experiment. Furthermore, the appendix contains a larger version of a sample JPDD which has been presented to the participants during the experiment.

Chapter 2

Overview Of Aspect-Orientation

This chapter introduces the key concepts of aspect-oriented software development in order to provide the reader with the knowledge that is needed to understand the contents of this thesis. Apart from the knowledge presented here, the reader is expected to be rudimentarily familiar with Java [*Arnold et al. (2000)*] and UML [*Booch et al. (1998)*].

2.1 The General Approach

2.1.1 Crosscutting Concerns

Aspect-oriented software development is a novel software development approach which aims at modularizing crosscutting concerns. **Crosscutting concerns**[1] are concerns whose implementation cannot be encapsulated into a single module with conventional software development methods. As a consequence, the implementation of a crosscutting concern spreads out across the entire application, i.e. it "crosscuts" all (or at least some) of the other modules of the application. The phenomenon of crosscutting is generally considered to obstruct the comprehension of the implementation of the crosscutting concern because it is spread out (or **scattered**) across the entire program. In addition to that, the scattered program code in the crosscut modules often looks very similar (or is identical, even), which makes initial specification and subsequent modification of the code a tedious and laborious task. Finally, crosscutting is also considered to blur the primary concern of the modules being crosscut because the module gets swamped (or **tangled**) with code from other ("non-primary") concerns. In summary, the resulting code organization is generally considered unfortunate because it impedes code comprehension and makes implementation and modification of the program code difficult.

2.1.2 Aspects

In order to overcome the problem mentioned above, aspect-oriented software development approaches offer a modularization mechanism which permits to encapsulate the implementation of a crosscutting concern in just one module. This modules is generally referred to as the **aspect**. The aspect defines the core functionality of the crosscutting concern as well as when and where in the other modules that crosscutting functionality shall be performed. The points in other modules where or when the crosscutting functionality shall be performed are referred to as **join points**[2]. Correspondingly, the

---

[1] The term "crosscut" was first used by [*Kiczales et al. (1997)*]; a conceptual formalization of the term can be found in [*van den Berg et al. (2007)*].

[2] The term "join point" was first used by [*Kiczales et al. (1997)*], where join points have been defined as "those elements of the component [i.e. base] language semantics that the aspect programs coordinate with". A less conceptual yet more practical definition of the term "join point" can be found in [*Hanenberg (2006)*], where – roughly speaking – join points are defined as "any element which can be selected and adapted by an aspect-oriented system".

definition of the (set of) join points at which the crosscutting functionality should be performed is referred to as **join point selection**. And the definition of the functionality that should be performed at those join points is referred to as **join point adaptation**.

2.1.3 Join Points

It is important to note that join points can be classified in two (orthogonal) ways (cf. [*Hanenberg (2006)*]): there are static and dynamic join points, and there are structural and behavioral join points (see Figure 2.1 for an illustration). Static join points refer to elements in the program code (e.g. a class definition or a method definition), whereas dynamic join points refer to elements and situations that occur at runtime (e.g. objects, object state, or runtime events such as a method call). Dynamic join points have **join point shadows** [*Masuhara et al. (2003)*], which are those elements in the program code which define the dynamic join point (i.e. the object), or whose execution lead to the occurrence of the dynamic join point (i.e. to the object state or to the runtime event). Structural join points refer to entities that reify a structural abstraction in the program language being used (e.g. a class, an object, or a member of that class or object). Behavioral join points refer to elements which represent a part of the program behavior (e.g. a program statement in a method body, or the execution of such statement).

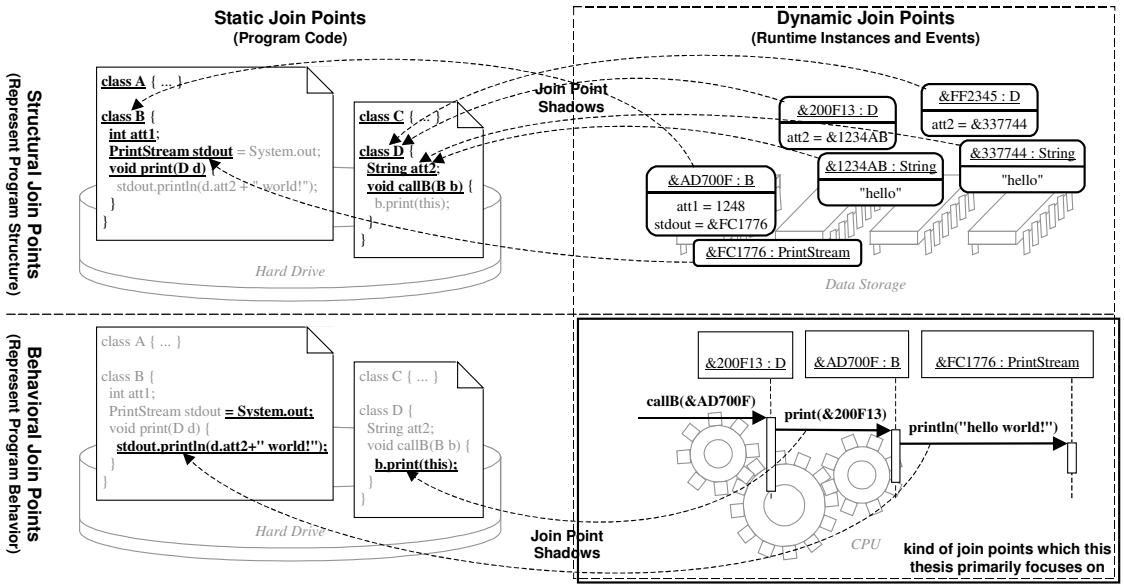


Figure 2.1 A classification of join points.

Different aspect-oriented systems usually support different kinds of join points. The particular kinds of join points which are offered by an aspect-oriented system – and which thus can be influenced by a crosscutting concern – are determined by the **join point model**[3] of the aspect-oriented system. Common examples of join points in such join point models are class and method definitions, constructor and method calls (and their executions), field references and field assignments (as well as their execution), etc. The

[3] The term "join point model" was first used by [*Kiczales et al. (2001)*, *Masuhara et al. (2003)*]. According to [*Masuhara et al. (2003)*], a "join point model" consists of join points, means to select these join points, and means to adapt these join points. The notion of a "join point model" used in this thesis stems from [*Hanenberg (2006)*], which considers join point selection means and join point adaptation means to be distinct from the join point model.

kinds of join points which this thesis primarily focuses on are dynamic behavioral join points, such as the occurrence of a method call at runtime.

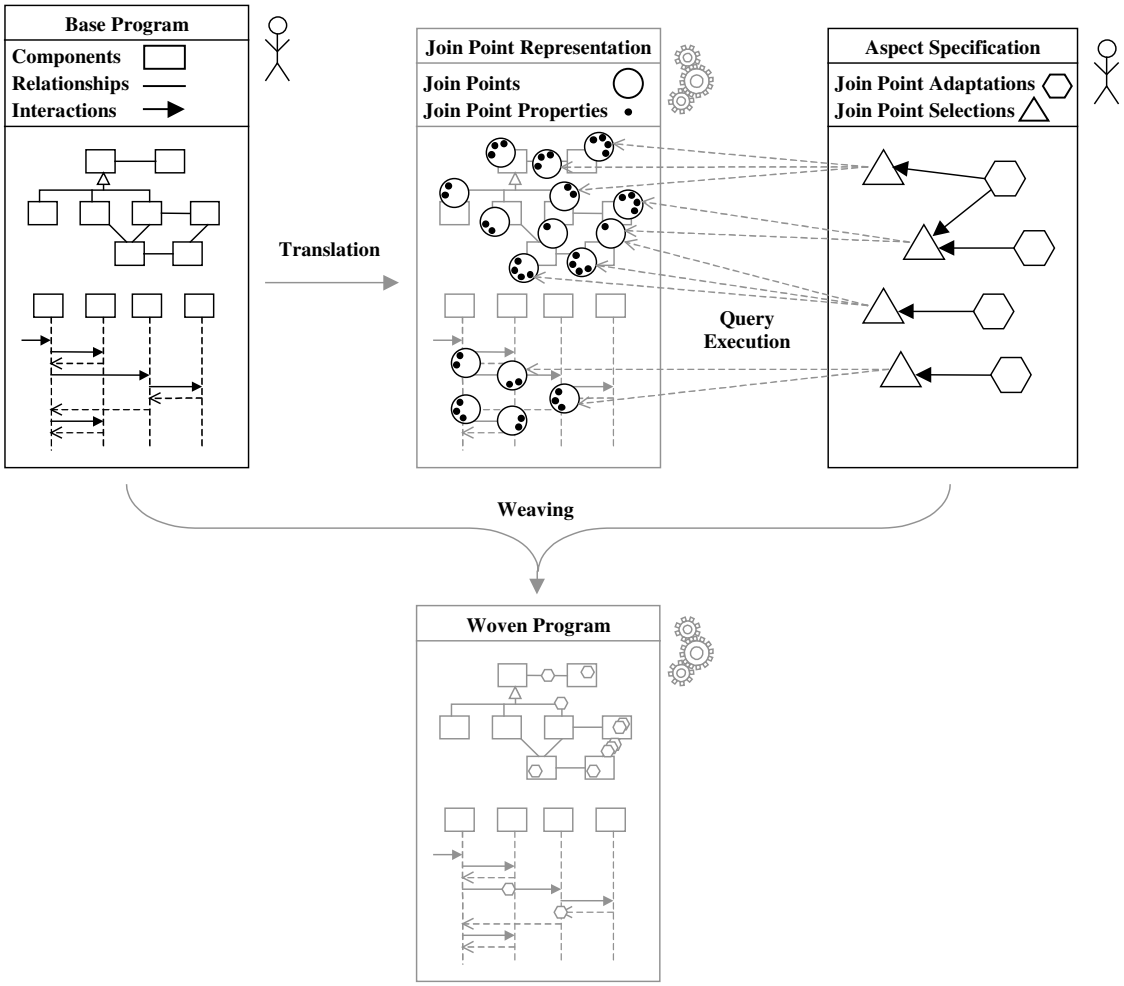


Figure 2.2 Ingredients of aspect-oriented systems (cf. [*Hanenberg (2006)*]).

2.1.4 Weaving

Figure 2.2 outlines the general procedure that is performed by an aspect-oriented system in order to implement the effects of a crosscutting concern into a given program (cf. [*Hanenberg (2006)*]). That process is generally referred to as **weaving** (cf. [*Kiczales et al. (1997)*]). Input to that procedure is the program code of the base program as well as the specification of the aspect(s). The aspect-oriented system takes the base program and computes a **join point representation** [*Kiczales et al. (1997)*, *Hanenberg (2006)*] for it. The join point representation contains all join points which can be selected and adapted by the aspects. Furthermore, it contains all properties of the join points which can be used to select and adapt them. Provided with the join point representation, the aspect-oriented system takes the join point selections from the aspect specification(s), and computes the sets of join points whose properties satisfy the selection constraints defined by the join point selections. Finally, the aspect-oriented system adapts the selected join points of each join point selection according to the specification of the join point adaptations that are associated with the respective join point selection. This final step leads to the *woven program*,

which contains all the crosscutting effects that have been specified by the aspect specifications.

2.1.5 Definition Paradox

A couple of concluding remarks on the aspect-oriented software development approach, at last. As stated and illustrated above, the overall objective of aspect-oriented software development approaches is to provide modularization and encapsulation means for crosscutting concerns. As an immediate consequence of this objective, aspect-orientation removes the crosscutting nature from the crosscutting concern in the sense that its program code is no longer crosscutting the program code of the base program. What remains, however, is a "crosscutting effect" of the aspect containing the program code of the (formerly crosscutting) concern. That is why this thesis keeps on using the term "crosscutting concerns" in order to refer to concerns that can be modularized and encapsulated by aspects.

2.1.6 Obliviousness

The specific characteristic of the "crosscutting effect" is that it takes place at points in the program code (or during the dynamic execution of the program) where (or when) the developer of the program have *not* explicitly called for it. This is what is commonly referred to as the **obliviousness** (cf. [*Filman & Friedman (2005)*]) of the base program[4]: it means that the base program does not explicitly invoke the behavior of an aspect, thus reducing the coupling of the base program and ideally making the base program independent of the aspect. The imposed manner in which crosscutting behavior is executed distinguishes it from conventional ways of executing behavior.

2.1.7 Join Point Addressing and Join Point Encoding

The kinds of effects that can be imposed on a base program (i.e. the join point adaptations that can be realized) depend on the capabilities of the aspect-oriented system. Likewise, the ways in which join points can be selected (i.e. the join point selection capabilities) vary between different aspect-oriented systems. In [*Hanenberg (2006)*], a systematic classification of those capabilities can be found. In that classification, join point selection capabilities are subdivided into **join point encoding** capabilities and **join point addressing** capabilities. Join point encoding capabilities relate to the kinds of join point properties which are provided by an aspect-oriented system in order to characterize and identify a join point. Join point addressing capabilities relate to the language constructs which are provided in order to specify selection constraints on those properties. With respect to this sub-distinction, this thesis can be considered to deal with – representational – issues of join point addressing, while simply presuming that all addressed join point properties are actually available (i.e. encoded). Still, this thesis keeps referring to join point selections as it is considered to be the more accessible term.

2.1.8 Classification of Join Point Properties

Similar to join points (see Figure 2.1), join point properties can be classified in two orthogonal ways, i.e. into structural and behavioral join point properties, as well as into

[4] Note that this does not necessarily imply that the developer, too, is "oblivious" to the presence of the aspect.

static and dynamic join point properties (see Table 2.1). Structural join point properties relate to reifications of structural abstractions of a program (examples are classes, objects, or the members of those classes or objects), whereas behavioral join point properties relate to execution steps of a program (examples are program statements in a method body, or the execution of such statements). Orthogonal to that, static join point properties refer to elements in the program code (examples are the name of a member in a class declaration, or the type of a parameter in a method declaration), whereas dynamic join point properties refer to elements and situations that are not known until runtime (examples are the type of an object, the value of an attribute, or the occurrence of previous runtime events). It is important to note that the kind of properties which are provided by an aspect-oriented system for a given join point (in its join point representations; see Figure 2.2) has an immediate impact on the classification of that join points. For example, a behavioral and dynamic join point must always have at least one behavioral and dynamic join point property – since otherwise it would not constitute a behavioral and dynamic join point. The reader should bear this constitutive relationship between the nature of join point properties and the nature of join points in mind while reading the remainder of this thesis. For a similar discussion, the reader is referred to [*Hanenberg (2006)*][5].

Table 2.1 A classification of join point properties (with examples).

	static	dynamic
structural	names of members in class declarations types of parameters in method declarations	values of attributes and associations between objects runtime types of arguments
behavioral	existence of preceding statements in method bodies (in program code)	occurrence of preceding method calls (at runtime)

2.1.9 On the Relevance of the Woven Program

Finally, it is important to note that the woven program is not to be seen nor to be contemplated by the developer. The process of weaving as well as its result is considered to be an "internal matter" of the aspect-oriented compiler – just as the realization of a lookup mechanism of a method call in an object-oriented programming language is considered an "internal matter" of the object-oriented compiler. There are aspect-oriented systems which do weaving "on the fly" or "on the meta-level", thus never producing a woven program which could be looked at and which could be analyzed (examples are the dynamic weaving approaches of PROSE [*Popovici et al. (2002)*, *Popovici et al. (2003)*], AspectS [*Hirschfeld (2002)*], or JAsCo [*Vanderperren & Suvee (2004)*], aspect-aware virtual machines such as STEAMLOOM [*Bockisch et al. (2004)*], or the load-time weaving approach found in JAC [*Pawlak et al. (2001)*][6]). Thus, rather than looking at the woven program code, developers are generally deemed to study the join point selections and join point adaptations of an aspect-oriented program in order to comprehend how a crosscutting concern impacts a given base program – similar to the way in which

[5] [*Hanenberg (2006)*] also makes a distinction between static and dynamic join point properties; however, no distinction is made between structural and behavioral join point properties; instead, a more fine-grained classification of join point properties is suggested (which is without relevance to the problems addressed by this thesis).

[6] which is meanwhile also supported by AspectJ [*Kiczales et al. (2001)*, *Laddad (2003)*]

developers inspect the inheritance hierarchy of an object-oriented program (rather than any compiled version of that program) in order to realize which method body will be (ultimately) executed when a method call is issued.

2.1.10 Focus of this Thesis

The reader should be informed that the focus of this thesis is on (the selection of) dynamic and behavioral join points. Consequently, the selection of static or structural join points and the adaptation of either kind of join points is not considered in the remainder of this thesis.

2.2 A Concrete Example: AspectJ

After having introduced the general concepts and mechanisms of aspect-oriented systems in the previous section, this section introduces a concrete example, i.e. the aspect-oriented programming language AspectJ [*Kiczales et al. (2001)*, *Laddad (2003)*]. The goal is to give the reader a principle idea of how the general concepts could be realized in a "real" working system.

AspectJ is probably the most popular aspect-oriented programming language with the largest user base. It is an extension to Java, and its (woven) byte-code can be run on conventional JVMs (Java Virtual Machines). Among the broad variety of currently existing aspect-oriented programming languages and systems, AspectJ has been chosen to be exemplified in this thesis due to its great popularity. This thesis refers to version 1.5.3 of AspectJ.

2.2.1 Anatomy of an Aspect

AspectJ introduces a new language construct, called "aspect", which encapsulate the program code of a crosscutting concern. AspectJ aspects are similar to Java classes in the respect that they may comprise fields and methods (which may be referred to and be invoked from somewhere else in the program code, provided that the visibility constraints[7] of the field or method definitions permit so). In addition to that, though, AspectJ aspects may contain so-called "advice" which determine the crosscutting effects that will be superimposed on a given program (i.e. an advice defines the join point adaptation). An advice has a parameter list which defines a list of formal variables which can be used in the body of the advice. These formal variables are bound to variables from the runtime context of the join point at which the advice takes effect. Each advice is affiliated to a so-called "pointcut" which determines the set of join points at which the advice shall be executed (i.e. a pointcut defines the join point selection). That pointcut also determines which variables from the runtime context of a join point are ultimately bound to the formal parameters of the advice. Pointcuts may be given a name and a signature, so that their pointcut definition can be used by more than one advice and even can be included in other pointcuts. In that case, pointcuts, too, have a parameter list which declares the formal parameters that are bound to variables from the runtime context of the join point.

[7] private, public, protected, or package protected

2.2.2 Example

The following code snippet (Listing 2.1) sketches the general anatomy of an aspect in AspectJ. The code snippet defines an aspect called "myAspect". That aspect has two conventional members: one private field of primitive type "int", named "myInt"; and one public method called "myVoidMethod()". Apart from that, the aspect has one pointcut, named "MyPointcut". The signature of pointcut "MyPointcut" comprises a parameter list with three formal parameters: a parameter "s" of type "String", as well as two parameters "i" and "j" of primitive type "int". The pointcut body is not shown here. At last, the aspect defines a (before[8]) advice with a parameter list consisting of three formal parameters: a parameter "t" of type "String", as well as two parameters "k" and "l" of primitive type "int". The advice is affiliated to the pointcut "myPointcut", which is eligible because types and numbers of formal parameters in the parameter lists of the advice and of the pointcut are compatible.

Listing 2.1 Anatomy of an aspect in AspectJ.

```
1  public aspect myAspect {
2
3    //members (may be private, protected, package protected, or public)
4    private int myInt;
5    public void myVoidMethod() {
6      //do nothing
7    }
8
9    //join point selection ("pointcut")
10   pointcut myPointcut(String s, int i, int j) :
11     //selection of the join points to crosscut
12   ;
13
14   //join point adaptation ("advice")
15   before(String t, int k, int l) : myPointcut(t, k, l) {
16     //implementation of the crosscutting effects
17   }
18
19 }
```

2.2.3 Join Point Selection (Pointcuts)

As stated above, a pointcut determines the set of join points at which the crosscutting effects of the aspect shall take place. To do so, the pointcut specifies a set of constraints which all selected join points must comply to. A pointcut body basically looks like a collection (conjunction and/or disjunction) of (possibly negated) Boolean functions. The "Boolean functions" are called primitive pointcut designators. Each of such pointcut designators constrains a particular characteristic of the join points that shall be selected. Most importantly, for example, pointcut designators may constrain the type(s) of join point that shall be selected. AspectJ permits to select field references (`get`) and field assignments (`set`), method and constructor calls (`call`), method and constructor executions (`execute`), the execution of class and object initializers (`staticinitialization` and `initialization`), the execution of an exception

[8] further explanations follow below.

handler (`handler`), and the execution of an advice (`adviceexecution`). Pointcut designators constraining the type of a join point divide the selectable join points in AspectJ into disjunctive subsets (note that this is not true for the other pointcut designators of AspectJ). Figure 2.3 illustrates the distinction of call and execution join points, for example (which may be designated by the `call` or the `execution` pointcut designator, respectively).

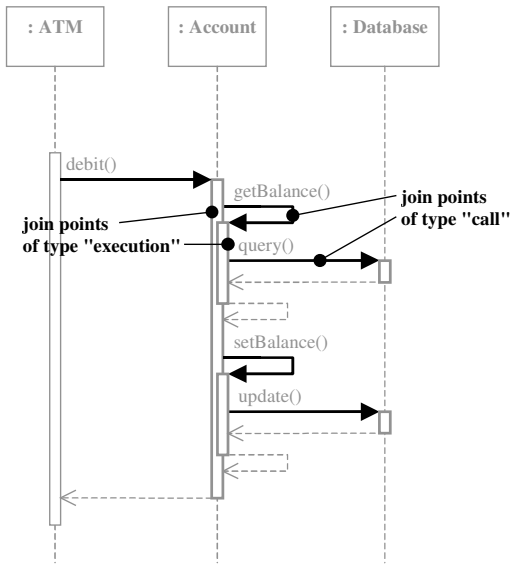


Figure 2.3 Call and execution join points in AspectJ.

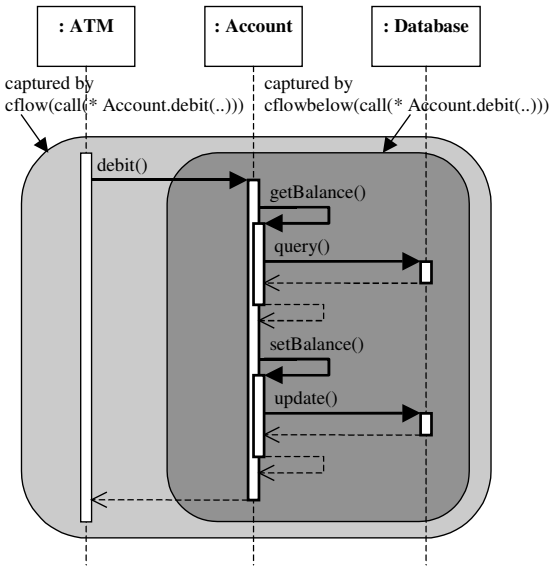


Figure 2.4 Join points designated by the pointcut designators `cflow` and `cflowbelow` (cf. [*Laddad (2003)*]).

Other pointcut designators of AspectJ may constrain other join point properties, such as the types of objects that exist in the runtime context of a join point (e.g. as the sender (`this`), the receiver (`target`), or an argument (`args`) of a method call). Or they may constrain the class definition (`within`) or the method definition (`withincode`) which the join point (or, in case of dynamic join points, their join point shadow) may occur in. The `if` pointcut designator verifies that an arbitrary Boolean condition (implemented in plain Java) is true for every selected join point. And finally, the `cflow` and `cflowbelow` pointcut designators require that all join points come to pass in the control flow of a particular other join point, such as a method call, for example (see Figure 2.4 for an illustration; cf. [*Laddad (2003)*]). An example elucidating some of these pointcut designators in closer detail are given below.

2.2.4 Join Point Adaptation (Advice)

As state above, pointcuts are affiliated to advice, which define the crosscutting effects that will be performed at the join points selected by the pointcut. The body of an advice in AspectJ looks like a conventional method body in Java. AspectJ permits to "advise" selected join points in three principle ways: `before` advice permit to execute supplementary behavior before the program code at the selected join point is executed; `after` advice permit to execute supplementary behavior after the program code at the

selected join point has been executed[9]; and `around` advice are executed in place of the program code at the selected join point[10]. The following Figure 2.5 illustrates these different ways to impact a join point (cf. [*Laddad (2003)*]):

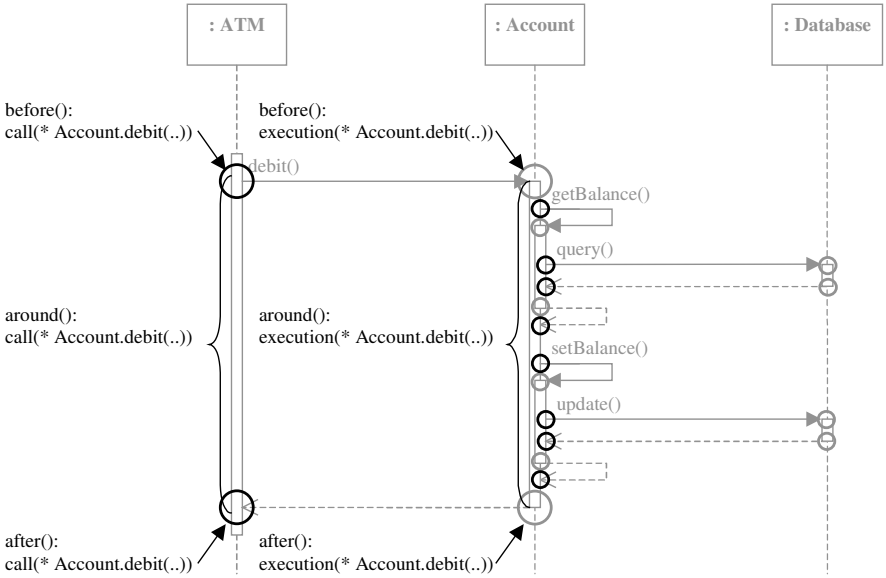


Figure 2.5 Points of advice execution in AspectJ (cf. [*Laddad (2003)*]).

2.2.5 Example

The following code snippet (Listing 2.2) shows a complete aspect definition, which exemplifies the implementation of a pointcut and an advice in closer detail. The example makes use of the pointcut designators `call`, `this`, `target`, `if`, and `cflow`. These pointcut designators are elucidated in further detail in the subsequent (together with the closely related pointcut designators `args`, and `cflowbelow`) because they are frequently used in those kinds of situations which this thesis is focusing on, i.e. the selection of dynamic and behavioral join points.

Listing 2.2 Sample aspect in AspectJ.

```
1  public aspect myAspect {
2
3    //join point selection ("pointcut")
4    pointcut myPointcut(YoungstersAccount acc, Database db) :
5      //selection of the join points to crosscut
6      cflow(call(* Account.debit(..)) && this(ATM))
7      && call(* Database.update(..)) && this(acc) && target(db)
8      && if(acc.getBalance() < acc.getCreditLine())
9    ;
10
11   //join point adaptation ("advice")
12   after(Account acc, Database db) : myPointcut(acc, db) {
```

[9] In case of `after` advice, the modifier `returning` or `throwing` may be used to make the advice take effect only upon successful or unsuccessful termination, respectively, of the program code at the selected join point.

[10] In case of `around` advice, the special keyword `proceed` may be used in order to invoke the (initially intercepted) behavior at the selected join point manually.

```
13      //implementation of the crosscutting effects
14      db.log(acc + " is overdrawn.");
15    }
16
17 }
```

2.2.6 call Pointcut Designator

The `call` pointcut designator selects all calls to a particular set of methods. In order to define these methods, the `call` pointcut designator expects a method pattern as argument. By means of this method pattern, the methods whose calls are going to be selected are further confined with respect to the name of the methods, the kinds of their modifiers (e.g. public, static, etc.), the name of the classes defining the methods, the name of the packages defining the classes, the number and the types of arguments, as well as the return types. All names and types in the method pattern may make use of an asterisk wildcard (`'*'`) in order to abstract over an arbitrary number of characters. A dot-dot-wildcard (`'..'`) may be used to abstract over an arbitrary number of levels in the package hierarchy, or to abstract over an arbitrary number of parameters in the parameter list. A plus operator (+) behind a type indicates that all subtypes of that type are to be considered as well.

In the example shown above, the call pointcut designator occurs twice. The first occurrence refers to all calls to methods whose name is "debit" and which are declared in class (or interface) "Account". The method may take any number of arguments (which may be of any types) and may return any type of return value (or none at all). The second occurrence refers to all calls to methods whose name is "update" and which are declared in class (or interface) "Database". Again, the method may take any number of arguments (of any types) and may return any type of return value (or none at all).

2.2.7 this, target, and args Pointcut Designators

The `this`, `target`, and `args` pointcut designators constrain the runtime types of the sender, receiver, and argument objects of a method call. Moreover, they may expose these objects to the pointcut and thus to the advice, so that they can be accessed by the pointcut and possibly modified by the advice. `This`, `target`, and `args` pointcut designators expect a type or an identifier as argument. If a type is used, the pointcut designator merely constrains the runtime type of the corresponding object (i.e. the sender, the receiver, or an argument of the selected method call, respectively). If an identifier is used, the pointcut designator also exposes the corresponding object to the pointcut and (ultimately) to the advice. In the latter case, the identifier needs to be defined in the parameter list of the pointcut as a formal parameter. `This`, `target`, and `args` pointcut designators must not use wildcards when defining a type or an identifier. However, a singular asterisk wildcard (`'*'`) as well as a dot-dot-wildcard (`'..'`) may be used by the `args` pointcut designator to constrain the number of arguments of a method call yet not (all of) its types.

In the example above, `this` and `target` pointcut designators are used to constrain the types of the objects which may/must participate in the calls to methods "debit" or "update", respectively. Accordingly, the calls to methods "debit" must be initiated by an object of type "ATM". Furthermore, the calls to methods "update" must be initiated by an object of type "YoungstersAccount" and must be addressed to an object of type

"Database". Note in the latter case that the `target` pointcut designator refers to another type than the declaring type of the intercepted method (i.e. the type which is mentioned in the `call` pointcut designator). This is a frequent approach to constrain the join point selection to a particular subtype of the declaring type, only. Note further how `this` and `target` pointcut designators are used to also expose the calling object and the called object. To do so, the pointcut designators make use of the identifiers "acc" and "db", which are declared as formal parameters in the pointcut signature, rather than referring to the types "Account" and "Database" directly.

2.2.8 if Pointcut Designator

The `if` pointcut designator permits to specify that an arbitrary condition must hold at each selected join point. Therefore, the `if` pointcut designator expects an arbitrary Java expression whose evaluation yields a Boolean value (i.e. true or false)[11]. The expression is evaluated at each join point, and the affiliated advice is executed only if the expression evaluates to true.

The example above makes use of a `if` pointcut designator to test if the current balance of the account is lower than the current creditline of the account. The test is performed prior to the execution of the advice. The advice executes only, if the condition in the `if` pointcut designator holds (i.e. if the return value of method "getBalance" of the "YoungstersAccount" object which initiated the "update" method call actually *is* lower than the return value of method "getCreditLine" of that same object).

2.2.9 cflow and cflowbelow Pointcut Designators

The `cflow` and `cflowbelow` pointcut designators permit to specify that all selected join points must occur in the control flow of a particular (set of) other join point(s). This means that the "other" join points must still be "active" (i.e. on the call stack) when the "selected" join points are reached. The `cflow` and `cflowbelow` pointcut designators expect a pointcut definition as argument. That pointcut definition determines the set of join points in whose control flow selected join points are permitted to occur. If the pointcut definition selects a method call join point, for example, the `cflow` and `cflowbelow` pointcut designators select all join points which come to pass in the control flow of that selected method call. The difference between the `cflow` and `cflowbelow` pointcut designators is that the `cflow` pointcut designators includes the join point (here: method call) which initiated the control flow, while the `cflowbelow` pointcut designators doesn't (see Figure 2.4 for an illustration).

In the example above, the `cflow` pointcut designators is used to constrain the control flow in which the method "update" must occur in. To do so, the `cflow` pointcut designators is attributed with a pointcut definition which selects calls to method "debit". In consequence, the pointcut selects only those calls to method "update" which come to pass in the control flow of method "debit".

[11] The `if` pointcut designator may only refer to (static) class members and not to (non-static) object members – apart of those object members which are exposed by the pointcut designators `this`, `target`, and `args`.

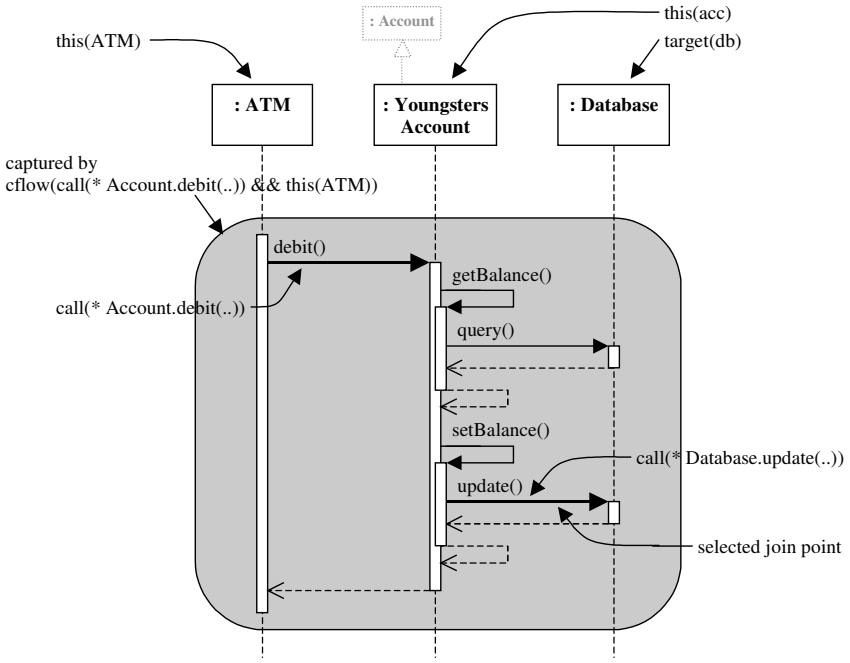


Figure 2.6 Sample scenario which leads to join point selection and join point adaptation by the sample aspect.

2.2.10 Summary of Example

In summary, the pointcut in the example shown in Listing 2.2 selects all calls to methods whose name is "update" and which are declared in class (or interface) "Database". The method may take any number of arguments (of any types) and may return any type of return value (or none at all). The calls must be initiated by an object of type "YoungstersAccount" and must be addressed to an object of type "Database". Both objects are exposed to the advice. Apart from that, the method calls must occur in the control flow of another method call, i.e. a call to some (other) method whose name is "debit" and which is declared in class (or interface) "Account". That (other) method may take any number of arguments (which may be of any types) and may return any type of return value (or none at all). And it must be initiated by an object of type "ATM". Last but not least, a method call satisfying all of these constraints leads to an execution of the advice only if the return value of method "getBalance" of the object which initiated the "update" method is lower than the return value of method "getCreditLine" of that same object. Figure 2.6 illustrates a scenario which would lead to a selection of a join point (provided that the constraint in the `if` pointcut designator holds; the constraint is not visualized in Figure 2.6).

2.2.11 Conclusions from the Example

It is important that the reader recognizes that many of the constraints specified by the pointcut shown in Listing 2.2 cannot be evaluated statically, i.e. at compile-time. This is a frequent issue when selecting behavioral and dynamic join points. Common examples of constraints that cannot be statically computed are constraints on the runtime type (e.g. that method "update" must be initiated by an instance of type "YoungstersAccount"), control flow constraints (e.g. that method "update" must occur in the control flow of method "debit"), as well as the `if` pointcut designator (i.e. that the current balance of the account is lower than the current creditline of the account). In order to guarantee that these

constraints are satisfied before the advice is executed, AspectJ inserts so-called **join point residues** [*Hilsdale & Hugunin (2004)*] or **join point checks** [*Hanenberg et al. (2004)*] into the base program code at appropriate places. These join point checks are evaluated at runtime. They verify if a "join point candidate", i.e. a system event which satisfies all statically computable constraints of the pointcut, actually represents a "real join point", and thus leads to the execution of the advice, or not.

Moreover, join point checks sometimes need to resort to information that occurred at an earlier point in runtime in order to be evaluated. One situation in which this is the case is a join point check that implements a control flow constraint, for example. In order to decide if a particular join point comes to pass in the control flow of a particular other join point, the join point check needs to acquire from somewhere if that other join point ever occurred (and still is active, i.e. has not suspended yet). To cope with that, AspectJ inserts code at the other join point (e.g. at method "debit") which keeps track of all occurrences and suspensions of the join point. The tracking information may be accessed by the join point check which is inserted at the join point of interest (e.g. at method "update"). And thus, the join point check may decide whether or not the advice needs to be executed.

2.2.12 Focus of this Thesis

It is important to note that the particular focus of this thesis is on such join point selections whose selection constraints pertain to information from an earlier point in runtime in order to be evaluated. The join point properties which are constrained by these kinds of join point selections can be classified as "dynamic and behavioral". That is, they denote properties which resort to the (previous) execution steps of a program and which cannot be determined before runtime. (Recall that this also implies that all join points which fulfill these constraints can be classified as "dynamic and behavioral", too; cf. section 2.1.)

2.2.13 Further Features

A couple of remarks on further features of AspectJ, at last. Apart from pointcuts and advice, AspectJ offers a means called "inter-type declaration" (formerly known as "introduction") which permits to extend classes with additional fields and methods and to augment the class hierarchy with supplementary inheritance relationships. Furthermore, it provides a special `declare precedence` notation which addresses the issue of aspect precedence and advice precedence (the issue arises whenever two advice in different aspects apply to the same join point; in such cases, it needs to be decided which of the two advice is executed first so that the software system shows the desired behavior[12]). Note that both of these features are beyond the scope of this thesis. This is because inter-type declaration always refer to structural and static join points (rather than dynamic and behavioral join points), and precedence declarations always pertain to join point adaptations (rather than join point selections).

Apart from that, AspectJ provides various means to instantiate aspects in different ways, e.g. for particular objects, for particular control flows, or as singletons (i.e. once for the entire system). An aspect may behave differently depending on the particular way in

[12] This problem is generally referred to as the problem of aspect interaction [*Douence et al. (2002)*, *Douence et al. (2004)*, *Mussbacher et al. (2008)*] or aspect interference [*Bergmans (2003)*, *Aksit et al. (2009)*].

which it is instantiated (especially if it maintains aspect-instance-specific state). Therefore, choosing the right aspect instantiation mechanism may sometimes be an intricate means to (further) constrain a join point selection. In that regard, the aspect instantiation means of AspectJ do relate to the focus of this thesis (which is the specification of join point selections). However, for the remainder of this thesis, choosing the right aspect instantiation mechanism to implement a join point selection properly is considered to be an implementation-specific wrinkle, and thus no distinction is made to other ways of implementing join point selections (e.g. with help of pointcuts and advice).

2.3 Summary

This chapter has described the new concepts and mechanisms that aspect-orientation brings about to modern software development in order to overcome the problem of crosscutting concerns. It has given an overview of the general procedure as well as of the key artifacts of the approach, and it has exemplified a concrete implementation of it (i.e. AspectJ). In particular, the chapter has introduced the notion of a "join point" which is the point in program code or in time (i.e. during the dynamic execution of the program code) where or when an aspect may alter a base program. The chapter has introduced the notion of a "join point property" which characterizes a join point in a particular respect, and which may be used to discriminate and select a join point. To do the latter, the chapter has introduced the notion of a "join point selection" which circumscribes a set of join points by specifying constrains on (their) join point properties. The chapter has sketched how selected join points may be adapted by means of "join point adaptations". It has been emphasized, however, that the focus of this thesis is on the selection of join points.

In the chapter, it has been recognized that join points can be classified in two ways, i.e. into dynamic and static join points, and into behavioral and structural join points. An analogous classification has been established for join point properties, and it has been presumed that a join point of a certain kind will always possess at least one join point property of the analogous kind.

It has been pointed out that the focus of the remainder of this thesis is on (the selection of) dynamic and behavioral join points. Join points of that kind come to pass during the dynamic execution of a program and refer to the (instant) execution of a program statement (such as a method call, for example). In order to designate such join points, join point selection particularly need to constraint the dynamic and behavioral properties of those join points. Dynamic and behavioral join point properties are properties whose values cannot be determined until runtime, e.g. because they constrain the past (i.e. earlier) program behavior (an example is the `cflow` pointcut designator in AspectJ).

2.3.1 Outlook to Next Chapter

The specification of join point selection constraints on dynamic and behavioral join point properties may become very complicated. It is subject of the next chapter to illustrate situations in which join point selections specifying selection constrains on dynamic and behavioral join point properties may impose problems on the software developer (especially, if the join point selections constrain earlier program behavior and resort to information that needs to be collected from different moments at runtime).

Chapter 3

Problem Statement

Sharing knowledge about program code is an important and recurring necessity in software development. Developers need to understand what their co-developers have implemented before they can deploy or evolve their program code. Apart from that, developers may learn from existing solutions and adapt them to solve similar problems. As history has shown, acquisition of good solutions may even work among developers if they are not using the same programming languages. The GoF design patterns [*Gamma et al. (1995)*] are a prominent example of such cross-fertilization.

Sharing knowledge is also important in aspect-oriented software development. One particular task of aspect-oriented software development is the implementation of join point selections, i.e. the determination of circumstances under which a piece of software shall be influenced by an aspect. For that purpose, aspect-oriented programming languages come with specific language constructs that permit to constrain individual characteristics of such situations. It is inevitable that developers need to understand the semantics of these language constructs before they can understand, deploy, or evolve the implementation of a join point selection.

Unfortunately, the implementation of a join point selection may quickly grow complex. Complex implementations of join point selections usually consist of multiple programming units. A prominent example are join point selections that resort to information which needs to be collected from different moments in the dynamic execution of a program: their implementation is usually built from multiple programming units where each one is responsible for gathering a different piece of relevant information about the running program[13]. The resulting complexity of such implementations represents a significant impediment to knowledge sharing among aspect-oriented developers. This is because developers need to mentally reconstruct the objectives of the join point selection by inspecting each programming unit and investigating how they work together. That is, developers need to identify all components that are involved in the implementation, and they need to recognize the interdependencies that exist between those components, and they are required to assess their impacts. This is a laborious and error-prone task.

The problem is even worse when developers are trying to read and understand an implementation of a join point selection written in another programming language. In that case, developers have to learn the programming language first before they can begin to investigate what the implementation is about. The extra burden of learning a new language, however, may prevent developers from looking at the solutions of other developers (using a different programming language), and thus may impede cross-fertilization between developer communities using different programming languages.

[13] A couple of aspect-oriented programming languages provide dedicated means which permit to specify particular join point selections of the mentioned kind in a single programming unit; however, the prevailing majority of aspect-oriented programming languages fails to do so, or provides only insufficient means (exemplary evidence is given later in this section).

To illustrate the problem, this section presents several complex implementations of join point selections implemented with different aspect-oriented programming languages. With help of these examples, the section discusses the problems that developers are faced with when they are trying to read and understand complex implementations of join point selections (possibly in a different programming language).

3.1 Impediments of Knowledge Sharing in Aspect-Oriented Babylon

As explained above, complex implementations of join point selections may lead to frequent comprehension and/or communication problems among software developers, and thus may impede the knowledge transfer among them. The goal of this section is to discuss in closer detail and with help of examples what intellectual efforts it takes to read and understand a complex implementation in order to recognize the objectives of the original join point selection, i.e. it pinpoints the assumed reasons to such comprehension and/or communication problems.

In order to do so, various implementations of join point selections are presented in various aspect-oriented programming languages. The examples illustrate how complex implementations of join point selections may look like in different aspect-oriented programming languages, and they illustrate the reasons why the implementations of such join point selections are so complex. Moreover, the examples emphasize what a developer must know about a particular programming language before he/she can read and understand a workaround implementation in that programming language. Further examples are presented in Appendix A.

3.1.1 Implementing a Sanitizing Aspect with AspectJ

The first example illustrates the implementation of an aspect with AspectJ [*Kiczales et al. (2001)*, *Laddad (2003)*]. AspectJ is probably the most popular aspect-oriented programming language. It is based on Java, and provides three new language constructs (amongst others, which are not relevant to this thesis): a **pointcut** defines the circumstances under which an aspect will impact the base application; an **advice** defines how the base application is going to be effected whenever those circumstances occur; and an **aspect**, finally, serves as a container for the former two. Aspects may contain ordinary Java fields and methods (similar to common Java classes). The following implementation makes use of this feature.

The following code snippet shows an implementation of a "sanitizing aspect" (see Listing 3.1). The example is adopted from [*Masuhara & Kawauchi (2003)*], and it is intended to fix an internet security hole which may be used by attackers to inject a malicious script into a dynamically generated web page; the script may then reveal confidential data as soon as it is executed on the client machine (the problem is also known as "cross-site-scripting"; cf. [*Masuhara & Kawauchi (2003)*]). In order to prevent this, the sanitizing aspect quotes, or "sanitizes", any data coming from the internet (such as HTTP request parameters, HTML form data, or cookies, for example) before the data is added to dynamically generated web pages. The example discussed in the subsequent is concerned with the sanitation of cookies.

3.1.1.1 Explanation of the Code

In order to implement the sanitation concern, the aspect defines three pointcuts, three advice, and two data stores (see Listing 3.1). The first pointcut `dataOrigin` monitors the origin of "unsafe" data, i.e. the retrieval of cookies which are sent along with a HTTP request (see lines 6-11). The pointcut is affiliated to an advice which adds the retrieved cookies to the `cookies` data store.

The second pointcut `dataAccess` monitors the usage of the "unsafe" data which leads to the production of further "unsafe" data (see lines 13-19). In this case, this is the retrieval of the cookie value. Note that cookie values are considered "unsafe" only if they are retrieved from "unsafe" cookies (i.e. if the cookie they are retrieved from is contained in the `cookies` data store; see `if`-constraint of pointcut). If this is the case, the advice affiliated to the pointcut adds the cookie value to the `values` data store.

The last pointcut `dataDisposal` selects the points in time when any "unsafe" data needs to be sanitized (see lines 21-27). In this case, this is when the cookie values are to be printed out to the dynamically generated webpage (using a `PrintWriter` object). Accordingly, the pointcut requires that the argument of the print method must be contained in the `values` data store of the aspect. Only if this is the case, the aspectual adaptation (i.e. the sanitation, which is implemented by the advice affiliated to pointcut `dataDisposal`) is performed.

Listing 3.1 Complex implementation of a join point selection in AspectJ.

```
1  public aspect Sanitizing {
2
3    private static Collection cookies = new HashSet();
4    private static Collection values = new HashSet();
5
6    private pointcut dataOrigin():
7      call(Cookie HttpServletRequest.getCookie(String));
8
9    after() returning (Cookie cookie) : dataOrigin() {
10     cookies.add(cookie);
11   }
12
13   private pointcut dataAccess(Cookie cookie):
14     call(String Cookie.getValue()) && target(cookie)
15     && if(cookies.contains(cookie));
16
17   after(Cookie cookie) returning (String value) : dataAccess(cookie) {
18     values.add(value);
19   }
20
21   public pointcut dataDisposal(String value):
22     call(* PrintWriter.print*(String)) && args(value)
23     && if(values.contains(value));
24
25   void around(String value) : dataDisposal(value) {
26     //aspectual adaptation
27   }
28 }
```

3.1.1.2 Discussion

In order to understand the AspectJ code in Listing 3.1, readers of the code need to know what a pointcut is and they need to understand the semantics of AspectJ's primitive pointcut designators (such as `call`, `target`, `args`, and `if`). Furthermore, they need to be familiar with the concept of advice and need to know of its different "flavors" (such as `before`, `around`, and `after`). Apart from that, though, readers of the code also need to be familiar with Java, i.e. with the Java Collection API in particular, since the aspect uses Java HashSets as a data storage in order to fulfill its task.

Even if readers are well familiar with AspectJ and Java, though, understanding the aspect code is no trivial task. This is because much of the aspect code is concerned with finding the right join points rather than with defining the actual effects that shall be performed at these join points: two out of three advice are concerned with the maintenance of the data structures of the aspect. These data structures are then used by two of the three pointcuts in order to determine whether they should select a method invocation or not (note that the data stores are not involved in the actual aspectual adaptations). This leads to implicit data dependencies between the pointcuts and the advice which are very hard to detect, and which require careful code inspection. This is all the more true because some of the relevant data is exposed and processed by the pointcuts (e.g. the target object and the argument) and some is exposed and processed by the advice (i.e. the return values). There is no programmatic construct which indicates the existence of the data dependencies[14], and no programmatic construct which suggests the order in which the designated methods have to occur. In consequence, the investigation of the code of this aspect is a tedious and exhausting task and may easily lead to a wrong or incomplete estimation of its effects.

3.1.2 Implementing a Safe Iterator Aspect with AspectC++

The next example is implemented in AspectC++ [*Spinczyk et al. (2002)*]. AspectC++ is an aspect-oriented extension to C++ and very similar to AspectJ. AspectC++, too, offers **aspects**, **advice**, and **pointcuts** to implement crosscutting behavior, and their semantic is similar to that of their counterparts in AspectJ. The example implementation presented in the following makes use of these language constructs.

The example implements a "safe iterator aspect" which traps all attempts to access an iterator of a vector which has been invalidated due to a modification of the vector. In C++, any insertion/deletion of a vector element is considered to invalidate the iterators of the vector[15] because the modification may cause a reallocation of the vector in memory (cf. [*ISO (1998)*]). After the modification, the iterators may therefore point to the wrong memory locations and may thus yield wrong values. The objective of the safe iterator aspect is to identify such invalid accesses to iterators in order to take appropriate steps which prevent the program from crashing or from producing undesired results. The example (and its implementation) is adopted from [*Allan et al. (2005)*].

[14] Note that different variable names could have been used in the pointcuts and in the advice without changing the semantic of the code snippet shown above, thus making the detection of the data dependencies even more difficult.

[15] Note that this does not need to be true for iterators pointing to elements which precede the modification point.

3.1.2.1 Explanation of the Code

In order to implement the safe iterator aspect, the example code[16] shown in Listing 3.2 defines three hash maps. The first one (`vec_state`) is used to log the last modification of a vector. The second one (`it_vec`) is used to collect all iterators together with the vectors they are pointing to. And the last one (`it_vec_state`) is used to store the state of a vector by the time an iterator is created. These three hash maps are maintained as follows.

The first hash map `vec_state` is maintained by pointcut `updateVector` and its affiliated advice (lines 10-17): the advice stores a new (or replaces an existing) state label for each vector in the hash map whenever a vector is modified (a state label is implemented as a mere pointer to an empty `DummyObject`).

The second and third hash maps are maintained by pointcut `createIterator` and its affiliated advice (lines 19-27): the advice updates the second hash map `it_vec` with a new iterator→vector mapping whenever an iterator is newly created from a vector. In addition to that, the advice copies the current state label of the vector from the first hash map `vec_state` to the third hash map `it_vec_state`.

Finally, pointcut `accessIterator` and its affiliated advice compare the state of the vector previously stored in the third hash map `it_vec_state` to the current state of the vector, which is stored in the first hash map `vec_state`, whenever the iterator is accessed (lines 29-41). If those two state labels differ, the aspect executes the join point adaptation (see lines 33-37). Otherwise, the aspect resumes with the originally intercepted program behavior (by calling `proceed` on AspectC++'s special "thisJoinPoint" object, `tjp`; see lines 38+39).

Listing 3.2 Complex implementation of a join point selection in AspectC++.

```
 1 aspect SafeIterator {
 2   private:
 3     map<MyCharVector*,DummyObject*> vec_state;
 4     map<MyCharIterator*,MyCharVector*> it_vec;
 5     map<MyCharIterator*,DummyObject*> it_vec_state;
 6     typedef map<MyCharIterator*,MyCharVector*>::iterator MapIterator;
 7
 8   public:
 9
10     pointcut updateVector(MyCharVector* vec) =
11       call("% MyCharVector::insert(...)" ||
12            "% MyCharVector::erase(...)") && target(vec);
13
14     advice updateVector(vec) : after (MyCharVector* vec) {
15       vec_state.erase(vec);
16       vec_state.insert(make_pair(vec, new DummyObject()));
17     };
18
```

[16] Note that support of template instance matching is still experimental in AspectC++; therefore, the following AspectC++ code has been tested with help of a (semantically equivalent) wrapper class which wraps `vector<char>` from the Standard Template Library (STL).

[17] AspectC++ provides a special `result` keyword which exposes the return value of a function; the keyword does not work with functions returning anything else but references or pointers, though.

```
19    pointcut createIterator(MyCharVector* vec) =
20      call("MyCharIterator MyCharVector::begin()" ||
21           "MyCharIterator MyCharVector::end()") && target(vec);
22
23    advice createIterator(vec) : after (MyCharVector* vec) {
24      MyCharIterator* it = tjp->result()[17];
25      it_vec.insert(make_pair(it, vec));
26      it_vec_state.insert(make_pair(it, vec_state.find(vec)->second));
27    };
28
29    pointcut accessIterator(MyCharIterator* it) =
30      execution("% MyCharIterator::operator*(...)")[18] && that(it);
31
32    advice accessIterator(it) : around (MyCharIterator* it) {
33      if (
34        vec_state.find(it_vec.find(it)->second)->second !=
35                            it_vec_state.find(it)->second
36        ) {
37        //join point adaptation
38      } else {
39        tjp->proceed();
40      }
41    };
```

Unfortunately, the three pairs of pointcuts and advice mentioned above are not enough to realize the desired behavior. This is due to two reasons: first, the methods which create the iterators return them "by value" rather than "by reference"; and second, AspectC++ may only expose "local copies" of return values, i.e. those copies which are local to the returning function. No means are available in AspectC++ to expose the (variable which holds the) return value in the calling context. As a result, the pointer to the return value which is stored to the hash maps `it_vec` and `it_vec_state` by pointcut `createIterator` (see line 24) is not of much use because it stems from the local context of the returning function and does not survive the context switch from the returning function to the calling function. Consequently, it can never be used by pointcut `accessIterator` and its affiliated advice to search for a corresponding vector and its state, and thus the join point adaptation will never occur.

To overcome this problem the aspect must observe the copying process during the context switch, i.e. when the return value in the context of the returning functions is copied to the context of the calling function. Subsequently, it must add the copied iterator to the second hash map `it_vec`, while the state label of the vector it is pointing to must be copied to the third hash map `it_vec_state`. To do so, the aspect defines another pair of pointcut and advice (named `copyIterator`; see lines 43-54). Note that the additions to both hash maps occur under the condition that the iterator being copied already exists in the hash map `it_vec`, i.e. that it has been added to hash map `it_vec` by advice `createIterator` at an earlier point in time; this is necessary in order to make the comparison of the vector state labels in advice `accessIterator` failsafe.

[18] AspectC++ does not allow to intercepts *calls* to the dereference operator "`*`"; that is why the pointcut `accessIterator` monitors the *execution* of the operator method, instead.

Pointcut `copyIterator` and its affiliated advice are still not enough, though, to make the aspect work as desired. This is because the aspect still fails whenever a newly created iterator is assigned to an *existing* iterator variable in the calling context. In this case, the return value of the returning function is copied to an *implicit* variable in the calling context using the copy constructor, first. And then, the value of that implicit variable is assigned to the existing iterator variable using the assignment operator "=". Hence, in order to get a hold of all relevant iterators in these situations, too, the aspect must intercept all usages of the assignment operator "=". This is accomplished by pointcut `assignIterator` and its affiliated advice (lines 56-68). Note that the code of the `assignIterator` advice is almost identical to the code of the `copyIterator` advice, except that the old values associated with the iterator must be deleted from the hash maps `it_vec` and `it_vec_state`, first; otherwise, the insertion of the new values would have no effect[19].

Last but not least, the aspect defines two pairs of pointcuts and advice for the purpose of housekeeping. The pointcuts `destructIterator` and `destructVector`, together with their affiliated advice (see lines 70-76 and 78-83, respectively) intercept all destructor executions of iterators and vectors, respectively. They ensure that the destroyed iterators and vectors are appropriately removed from the hash maps such that no unneeded memory remains allocated.

Listing 3.2 (continued)

```
42
43    pointcut copyIterator(MyCharIterator* lhs) =
44    construction("MyCharIterator") && args("const MyCharIterator&") && that(lhs);
45
46    advice copyIterator(lhs) : after(MyCharIterator* lhs) {
47      //similar code as in advice assignIterator(lhs)
48      MyCharIterator* rhs = static_cast<MyCharIterator*>(tjp->arg(0));
49      MapIterator mapIt = it_vec.find(rhs);
50      if (mapIt != it_vec.end()) {
51        it_vec.insert(make_pair(lhs, mapIt->second));
52        it_vec_state.insert(make_pair(lhs, it_vec_state.find(rhs)->second));
53      }
54    };
55
56    pointcut assignIterator(MyCharIterator* lhs) =
57      call("% MyCharIterator::operator=(const MyCharIterator&)") && target(lhs);
58
59    advice assignIterator(lhs) : after(MyCharIterator* lhs) {
60      MyCharIterator* rhs = static_cast<MyCharIterator*>(tjp->arg(0));
61      MapIterator mapIt = it_vec.find(rhs);
62      if (mapIt != it_vec.end()) {
63        it_vec.erase(lhs);
64        it_vec_state.erase(lhs);
65        it_vec.insert(make_pair(lhs, mapIt->second));
66        it_vec_state.insert(make_pair(lhs, it_vec_state.find(rhs)->second));
67      }
68    };
69
```

[19] Note that the `insert` function of a STL `vector<char>` does not overwrite existing values.

```
70      pointcut destructIterator(MyCharIterator* it) =
71        destruction("MyCharIterator") && that(it);
72
73      advice destructIterator(it) : after(MyCharIterator* it) {
74        it_vec.erase(it);
75        it_vec_state.erase(it);
76      };
77
78      pointcut destructVector(MyCharVector* vec) =
79        destruction("MyCharVector") && that(vec);
80
81      advice destructVector(vec) : after(MyCharVector* vec) {
82        vec_state.erase(vec);
83      };
84  };
```

3.1.2.2 Discussion

The AspectC++ code in Listing 3.2 illustrates impressively to what extent the readers of an aspect code may have to cope with language-specific peculiarities: more than half (i.e. four out of seven pairs) of pointcuts and advice are concerned with overcoming the "bare" data management of C++ (something that Java or Smalltalk users usually do not have to think about). The necessity to cope with context switches, variable assignments, and memory deallocations adds a significant amount of extra complexity to the aspect code.

Consequently, it is a great challenge for the reader to identify the one pair of pointcut and advice which actually implements the aspectual adaptation (i.e. pointcut `accessIterator` and its affiliate advice), and it is not at all easy to recognize that the execution of the aspectual adaptation depends on the data manipulations which are performed by the other advice (let alone *how* it depends on them). Thus, it is no trivial task to identify the order in which the various pairs of pointcut and advice must occur. Developers are likely to spend a long time until they observe that the aspectual join point adaptation performed by pointcut `accessIterator` and its affiliated advice will only occur if pointcut `createIterator` and its affiliated advice has been executed before (since otherwise no iterator would be present in the data stores) *and* if pointcut `updateVector` and its affiliated advice has been executed since then (since otherwise the state labels of the vectors will match, and the `if`-statement guarding the join point adaptation will fail). It will take another while until the developers recognize that the execution of pointcuts `copyIterator` and `assignIterator` and their affiliated advice is optional, and that the execution of pointcut `destructIterator` and `destructVector` and their affiliated advice is prohibitive.

Another issue for developers who are unfamiliar with C++ may be the utilization and handling of the data collection templates and of their iterators from the Standard Template Library (STL), such as the `map<,>` template and its iterator `map<,>::iterator`, in this case. A particular cause to comprehension problems, for example, may be the way to retrieve values associated with particular keys using the `find` method (of the map) and the `second` method (of the iterator). Another issue may be the way to add key-value pairs to the map with help of the `make_pair` function. Finally, developers need to know that the `insert` function of a map does not overwrite existing values (unlike the `put` method of a HashMap in Java, for example). In summary, developers are forced to memorize the

interface of the collection templates first, together with the semantics of their functions, before they can properly estimate how the data storage is manipulated and referenced.

3.1.3 Implementing a File Access Management Aspect with AspectCOBOL

This section investigates an aspect implementation in AspectCOBOL [*Lämmel & Schutter (2005)*]. AspectCOBOL is an aspect-oriented transformation framework which permits to weave aspects to standard COBOL programs. Aspects in AspectCOBOL are defined as **declaratives**, which is a standard COBOL language construct. They consist of a pointcut block and a subsequent advice block. To define both, AspectCOBOL complements the set of standard COBOL language constructs which may be used within these declaratives with a set of extra language constructs. In particular, these language constructs permit to implement name-based join point selections, context exposure, as well as before, after, around, and throw advice.

The example considered in the following is adopted from [*Lämmel & Schutter (2005)*], and implements a file access management aspect which intercepts file accesses to unopened files, e.g. in order to open the file first and then resume with the intercepted behavior.

3.1.3.1 Explanation of the Code

The implementation shown in Listing 3.3 starts out with a definition of the data storage which is needed by the aspect (see lines 6-18). Key to the data storage is a dynamic table (`DYNAMIC-TABLE`; lines 7-14) which may hold up to 999 entries (`CHECKOPEN-ENTRY`; see lines 8-14), each consisting of a ten-digit file reference (`CHECKOPEN-IDREF`; line 11) and a Boolean flag denoting the current opened/closed status of the file (`CHECKOPEN-STATE`; lines 12-14). The entries of the dynamic table may be accessed using an index variable (`CHECKOPEN-IDX`; see line 10). The size of the dynamic table may be dynamically increased by incrementing the variable `CHECKOPEN-MAX` (see lines 6+9). A Boolean helper variable (`END-OF-TABLE`; lines 15-17) is needed to report unsuccessful searches in the dynamic table. Another helper variable (`VAR-IDREF`; line 18) is needed to expose a reference to the file which is currently being accessed to the aspectual adaptation (i.e. the advice block).

Afterwards, the aspect implementation defines a couple of pointcuts and advice which operate on the data storage. The first pair of pointcut and advice is concerned with registering opened files to the dynamic table (which may mean to add new entries to the dynamic table; see `MY-OPEN-CONCERN` in lines 23-36), while the second pair of pointcut and advice is responsible for registering closed files to the dynamic table (see `MY-CLOSE-CONCERN` in lines 38-49). The third pair of pointcut and advice, finally, performs the actual join point adaptation (see `MY-ACCESS-CONCERN` in lines 51-63) – provided that the accessed file could not be found in the dynamic table (`IF AT-END-OF-TABLE` in line 61) or that its opened/closed status is set to closed (`OR CHECKOPEN-CLOSE (CHECKOPEN-IDX)` in line 61). Note how all advice make use of the index variable `CHECKOPEN-IDX` and the Boolean helper variable `END-OF-TABLE` in order to search for the exposed file reference `VAR-IDREF` in the dynamic table (i.e. in all `CHECKOPEN-ENTRIES`; see lines 27-31, for example) before they perform their actions[20].

[20] Note that the `SEARCH` code could be extracted to a special subroutine.

Listing 3.3 Complex implementation of a join point selection in AspectCOBOL.

```
IDENTIFICATION DIVISION.
 ASPECT-ID. FileAccessAspect.

DATA DIVISION.
 WORKING-STORAGE SECTION.
 01 CHECKOPEN-MAX       PIC 999 VALUE ZERO.
 01 DYNAMIC-TABLE.
  02 CHECKOPEN-ENTRY    OCCURS 999 TIMES
                        DEPENDING ON CHECKOPEN-MAX
                        INDEXED BY CHECKOPEN-IDX.
    05 CHECKOPEN-IDREF PIC 9(10).
    05 CHECKOPEN-STATE PIC 9   VALUE 0.
       88 CHECKOPEN-OPEN       VALUE 1.
       88 CHECKOPEN-CLOSED     VALUE 0.
 01 END-OF-TABLE        PIC 9   VALUE 0.
    88 AT-END-OF-TABLE          VALUE 1.
    88 NOT-AT-END-OF-TABLE      VALUE 0.
 01 VAR-IDREF           PIC 9(10).

PROCEDURE DIVISION.
 DECLARATIVES.

  MY-OPEN-CONCERN.
    USE BEFORE OPEN
    AND BIND VAR-IDREF TO IDREF OF FILE.
  MY-OPEN-ADVICE.
    SET CHECKOPEN-IDX TO 1.
    SEARCH CHECKOPEN-ENTRY
      AT END SET AT-END-OF-TABLE TO TRUE
      WHEN VAR-IDREF = CHECKOPEN-IDREF (CHECKOPEN-IDX)
        SET NOT-AT-END-OF-TABLE TO TRUE.
    IF AT-END-OF-TABLE
      ADD 1 TO CHECKOPEN-MAX
      MOVE VAR-IDREF TO CHECKOPEN-IDREF (CHECKOPEN-IDX)
    END-IF.
    SET CHECKOPEN-OPEN (CHECKOPEN-IDX) TO TRUE.

  MY-CLOSE-CONCERN.
    USE BEFORE CLOSE
    BIND VAR-IDREF TO IDREF OF FILE.
  MY-CLOSE-ADVICE.
    SET CHECKOPEN-IDX TO 1.
    SEARCH CHECKOPEN-ENTRY
      AT END SET AT-END-OF-TABLE TO TRUE
      WHEN VAR-IDREF = CHECKOPEN-IDREF (CHECKOPEN-IDX)
        SET NOT-AT-END-OF-TABLE TO TRUE.
    IF NOT-AT-END-OF-TABLE
      SET CHECKOPEN-CLOSED (CHECKOPEN-IDX) TO TRUE
    END-IF.

  MY-ACCESS-CONCERN.
    USE BEFORE
    (READ OR REWRITE OR WRITE OR DELETE OR START)
    AND BIND VAR-IDREF TO IDREF OF FILE.
```

```
55    MY-ACCESS-ADVICE.
56      SET CHECKOPEN-IDX TO 1.
57      SEARCH CHECKOPEN-ENTRY
58        AT END SET AT-END-OF-TABLE TO TRUE
59        WHEN VAR-IDREF = CHECKOPEN-IDREF (CHECKOPEN-IDX)
60          SET NOT-AT-END-OF-TABLE TO TRUE.
61      IF AT-END-OF-TABLE OR CHECKOPEN-CLOSED (CHECKOPEN-IDX)
62 *      PERFORM JOIN-POINT-ADAPTATION
63      END-IF.
64
65   END DECLARATIVES.
66
67  END PROGRAM FileAccessAspect.
```

3.1.3.2 Discussion

The AspectCOBOL example shown in Listing 3.3 illustrates the problems that developers may be faced with when they need to cope with an aspect-oriented programming language whose underlying concepts they are not used to. This means that, in order to understand the COBOL code shown above, developers need to think "the COBOL way": while the language constructs themselves are basically plain English, the underlying concepts require a different perception of program execution than what is commonly taught today. For example, developers need to think in terms of subroutines rather than methods, they need to think of structures instead of classes, they need to think of a global data storage and need to (temporarily) forget about modern achievements of software engineering such as information hiding and data encapsulation, just like they need to think of a fixed data storage rather than a dynamically allocated one[21].

Next, developers need to understand the "particular" way of defining data structures which are used by the aspects to store relevant data (such as the opened/closed state of files, in this case). In particular, developers need to understand the meaning of "levels" which are used to aggregate multiple data elements into complex data structures (examples are levels `01`, `02`, and `05` in lines 7, 8 and 11). Furthermore, they need to understand how tables (or arrays) of such complex data structures may be specified (see line 8, for example). Finally, they need to understand the way in which COBOL defines Boolean variables (which are represented as enumerations of the values "0" and "1", only one of which can be assigned to the corresponding data element at the next higher level; see lines 15-17, for example).

Apart from the data definition, the usage of the data storage in the aspect code may cause comprehension problems, too. In particular, developers need to understand how elements from tables (or arrays) consisting of complex data structures may be searched (see lines 27-31) and accessed (i.e. read or written; see lines 30, 34, and 36, for example). Likewise, developers need to get familiar with the (un)conventional way of assigning and referencing Boolean variables (see lines 44, 46, and 47, for example).

Both data definition as well as data usage "feels" very different from what developers are used to in younger programming languages. Nevertheless, getting them right is essential

[21] Note that the current COBOL 2002 standard comprises object-oriented constructs (which have been supported by COBOL compilers as early as the mid 1990ies); AspectCOBOL does not make use of them, though (cf. [*Lämmel & Schutter (2005)*]).

in order to identify the dependencies that exist between the different pieces of advice in the aspect. Developers need to understand when an advice just reads the data and when it modifies the data. Furthermore, they need to recognize when a data modification leads to a side effect that is going to impact the behavior of the other advice. This comprehension task is intricate because the aspect code does not contain any explicit hint on how the pieces of advice depend on each other and in which order they must occur (or must not occur). Hence, developers need to inspect the aspect code very carefully in order to find out about these interdependencies themselves, before they are finally able to mentally reconstruct the overall behavior and the purpose of the aspect.

3.1.4 Implementing a Contextual Logging Aspect with Alpha

Alpha [*Ostermann et al. (2005)*] is an academic programming language which permits to specify aspects whose join point selections may resort to pretty much any (runtime) information of an executing program. In particular, Alpha join point selections may refer to the abstract syntax tree, the execution trace, the heap, as well as the static type assignments of a program. Alpha aspects may intercept programs written in a language called "L2", which is a simple object-oriented toy language in the style of Java. Alpha implements an extension to that language which permits developers to specify their join point selections in terms of PROLOG rules. Advice may hook on to these join point selections and specify the join point adaptation in standard L2 program code.

The example shown here is adopted from [*Allan et al. (2005)*], and is about logging of user queries that are issued on a public information terminal. The idea is to log queries only when users are logged in (e.g. in order to provide the users with a history of their queries); queries of anonymous users shall not be logged.

3.1.4.1 Explanation of the Code

Alpha does not provide a special language constructs for aspects. Instead, pieces of advice may be incorporated right into classes (such as the `main` class) where they can be deployed to a given program block using the `deploy` language construct (e.g. in the `main` method). The Alpha class shown in Listing 3.4, for example, specifies a `before` advice (see lines 3-5) which hooks onto a join point selection called `methodsToLog` (which is explained below) and deploys this advice to a program block which executes method `run` of a class `baseprogram` (see lines 8-10).

Listing 3.4 Implementation of an aspect in Alpha.

```
 1 class main extds Object {
 2
 3   before methodsToLog(T,USER,QUERY) {
 4     join point adaptation
 5   }
 6
 7   bool main(bool x) {
 8     deploy(this) {
 9       (new baseprogram).run(true)
10     }
11   }
12 }
```

The join point selection `methodsToLog` is defined in another (PROLOG) file and consists of three rules (see Listing 3.5). The rules make use of various predicates (`calls`, `classof`, `now`, `before`, `or`, and `mostRecent`), which are provided (i.e. predefined) by the Alpha interpreter. Each rule defines a different scenario in which the invocation of a query shall be logged.

The first rule defines the "default" constraint that all selected join points must satisfy (see lines 1-8): it defines that an invocation of a query (`Q`) shall only be selected if a user is logged in at an information terminal (`T`). That is, the query (`Q`) must occur after the user has logged in (`I`) and before the user has quit or has logged out again (`O`)[22].

The second rule defines an alternative to the first rule (see lines 10-17). It refers to situations in which a user logs in (`I`) at a freshly (re)booted information terminal (`T`) where no user has ever quit or logged out yet. Correspondingly, the aspect should keep logging the user queries (`Q`) until the user quits or logs out. Note how the rule makes use of the PROLOG operator "`\+`" (negation by failure) in order to specify that no call to method "logout" or "quit" must exist in the execution trace (see lines 15+16).

Finally, the third rule refers to a situation in which a (i.e. the next) user logs in at an information terminal after another (i.e. the previous) user has logged out (see lines 19-27). To do so, the rule refers to the *most recent* calls of methods "login", "logout", and "quit" (see lines 22+25): the rule defines that queries (`Q`) are only selected if they occur after both the most recent login (`I`) and the most recent logout (`O`), while the latter must precede the former (see line 26). Note that the third rule contains an extra `calls` predicate (line 23) which is required in order to expose the argument of the call to method "login"[23].

Listing 3.5 Complex implementation of a join point selection in Alpha.

```
 1 methodsToLog(T,USER,QUERY) :-
 2   calls(Q,_,T,query,QUERY),
 3   now(Q),
 4   calls(I,_,T,login,USER),
 5   before(I,Q),
 6   or(calls(O,_,T,logout,_),calls(O,_,T,quit,_)),
 7   before(Q,O),
 8   classof(T,infoterminal).
 9
10 methodsToLog(T,USER,QUERY) :-
11   calls(Q,_,T,query,QUERY),
12   now(Q),
13   calls(I,_,T,login,USER),
14   before(I,Q),
15   \+calls(_,_,T,logout,_),
16   \+calls(_,_,T,quit,_),
17   classof(T,infoterminal).
18
19 methodsToLog(T,USER,QUERY) :-
20   calls(Q,_,T,query,QUERY),
```

[22] Note that this rule will never match because it refers to the future of program execution (..., `now(Q)`, ..., `before(Q,O)`); nevertheless, it is necessary because it makes Alpha observe and log the execution of the four relevant methods.

[23] Note that the `mostRecent` predicate could be certainly rephrased such that it supports context exposure; in this example, though, only the standard Alpha predicates shall be used (as they are defined in [*Ostermann et al. (2005)*]).

```
21   now(Q),
22   mostRecent(I,calls(I,_,T,login,_)),
23   calls(I,_,_,_,USER),
24   before(I,Q),
25   mostRecent(O,or(calls(O,_,T,logout,_),calls(O,_,T,quit,_))),
26   before(O,I),
27   classof(T,infoterminal).
```

3.1.4.2 Discussion

The program code shown in Listing 3.5 exemplifies nicely how the predefined PROLOG predicates of Alpha permit developers to explicitly disclose the dependencies that must exist between different method invocations. They permit to specify in which order the relevant method invocations must occur (or must not occur). Furthermore, they permit to explicate the data dependencies that must be satisfied by the method calls. As such, the use of PROLOG predicates improves much over contemporary solutions to specify join point selections in other programming languages.

However, in order to understand the join point query, readers must be familiar with logic programming, and in particular with PROLOG. Developers need to know that a comma-separated list of predicates represents a logic conjunction (AND) of those predicates, and that multiple definitions of a single rule denote a logic disjunction (OR) of those definitions. They need to be aware of the mechanism of "logic unification" of variables, i.e. the mechanism of finding and binding a (free) variable to a value from the fact base which satisfies all predicates where the variable is involved. Yet, before they can reason about (the results of) unification, they need to be able to distinguish a variable (starting with a capital letter) from a constant (starting with a lower-case letter). They need to know about the "anonymous variable" (rendered by an underscore "_") whose occurrences – in contrast to ordinary variables – are read and treated as distinct variables. And finally, they need to learn the meaning of the "\+" operator which is used to negate the result of a predicate.

It is not before developers have acquired these PROLOG basics, that they can set out and try to investigate the different scenarios in which a join point selection will occur. Thanks to the dedicated predicates in Alpha, this task may be less intricate than in other aspect-oriented programming languages because all dependencies between relevant method invocations have been made explicit. Nevertheless, though, the sheer number of predicates and the multiple (partly similar) definitions of the join point selection rule mean a significant complexity that readers of the program code need to master. To do so, they need to carefully read through each of the rules in order to mentally reconstruct the timely manner in which the methods have to be executed (which may not comply to the spatial order in which the methods are specified in the program code, like in Listing 3.5). While doing so, they need to pay careful attention to the variables which are used in multiple predicates in order not to overlook any important data dependency. Otherwise, the developers are about to misconceive the actual goal of the aspect.

3.2 Summary

The goal of this chapter has been to illustrate the problems that developers are faced with when they need to deal with complex implementations of join point selections. The

complex implementations usually consist of several programming units which interdependent on each other in various ways. Each of the programming units usually monitors a different program event which must occur in a particular order in order to lead to a join point. Additionally, the programming units may designate data from the monitored program events which is needed in order to perform the join point adaptation or which is required to be involved in another (later) event, too.

Identifying and comprehending the chronological dependencies and the data dependencies between the different programming units is a laborious and time-consuming task. Readers have to perform considerable intellectual efforts in order to study all components of the implementation and to comprehend their collaborative effect on the base program. They need to be highly attentive and concentrated so that they do not get confused by the sheer number of components and interdependencies between them.

Another problem arises from the fact that the program code of complex pointcut implementations looks very different in different aspect-oriented programming language. In parts, this problem is due to the fact that complex pointcut implementations usually make extensive use of the base programming language (i.e. the programming language which the aspect-oriented programming language is based on, such as Java in the case of AspectJ, C++ in the case of AspectC++, or COBOL in the case of AspectCOBOL, etc.). This means that readers usually need to acquire (at least a minimal set of) the key concepts that are implemented by the base programming language before they can set out to analyze and comprehend the pointcut implementation at all. This makes cross-fertilization between different communities of software developers very difficult.

Hence, the problem that developers are faced with when they want to share their knowledge about a complex pointcut implementation with another developer is twofold:

- Comprehension of the interdependencies that exist between the components of a complex pointcut implementation.
- Comprehension of the language constructs (i.e. the programming language) which is used to implement (the components of) the pointcut implementation.

Both issues – even on their own – require extensive and thorough investigation of the reader who wants to understand the complex implementation of a join point selection. This means, even if the reader is well familiar with the language constructs used to implement the join point selection, he/she finds him/herself opposed to the problem of discovering the interdependencies that exist between its components. And even if the pointcut implementation is fairly simple and does not contain much interdependencies, the overall objectives of the implemented join point selection may be difficult to conceive for someone who is unfamiliar with the language constructs used to implement it.

3.2.1 Outlook to Next Chapters

It is the goal of this thesis to explore a possible solution to both of the problems mentioned above. To do so, the following chapter investigates existing work which – directly or indirectly – aims to improve the situation of the developer outlined above. The chapter explains and illustrates why the existing approaches are insufficient to solve the mentioned problems, and why they thus leave developers stranded with their comprehension problems. In response to that, a new approach is presented in Chapter 5 and discussed in Chapter 6 which is deemed to improve the current situation.

Chapter 4

State Of The Art

Facilitating the comprehension of program code is one of the driving forces of aspect-oriented software development. The principal idea is that concerns are easier to understand when they are implemented in a localized (i.e. non-scattered) and isolated (i.e. non-tangled) way. However, the previous chapter has demonstrated that even localized and isolated implementations of (crosscutting) concerns give rise to comprehension problems – in particular when they comprise an implementation of a complex join point selection.

This section addresses existing work which directly or indirectly addresses the comprehension problem of complex implementations of join point selections, and discusses why these existing approaches are still problematic. It starts out with a general discussion about current programming languages. Afterwards, it focuses on existing visualization techniques and modeling languages.

4.1 Aspect-Oriented Programming Languages

4.1.1 Overview of Existing Approaches

Specifying join point selections is a pivotal task in aspect-oriented software development. As a result, aspect-oriented programming languages provide manifold join point selection means of varying sophistication. Much research has been accomplished in finding appropriate language constructs that permit developers to express all relevant characteristics of the selected join points in an explicit, yet concise way. For the sake of celerity of coding and comprehensibility of the resulting program code, developers should not need to implement a join point selection constraint manually any longer. Seeking to reach this goal, research has particularly focused on the provision of language constructs that permit to specify constrains on the dynamic and behavioral properties of join points. These kinds of properties most commonly refer to facts which occurred at an earlier point during the execution of a program. As a result, a variety of different language constructs has been developed. Examples are the `cflow` pointcut designator in AspectJ (see section 3.1.1), the `before` and `mostRecent` predicates in Alpha (see section 3.1.4), the data flow (`dflow`) pointcut designator presented in [*Masuhara & Kawauchi (2003)*] and implemented by [*Alhadidi et al. (2009)*], the Declarative Event Patterns (DEP) introduced by [*Walker & Viggers (2004)*], the Tracematch extension [*Allan et al. (2005)*] of abc [*Avgustinov et al. (2005)*], the ability to define Stateful Aspects [*Douence et al. (2004)*], e.g. with JAsCo [*Vanderperren et al. (2005)*], etc. Code snippets which exemplify the use of some of those language constructs are presented in Chapter 3 and Appendix A.

Each of these language constructs permits developers to constrain particular characteristics of the (earlier) runtime. For example, the language constructs of JAsCo [*Vanderperren et al. (2005)*], Tracematches [*Allan et al. (2005)*], and Alpha [*Ostermann*

et al. (2005)] permit to constrain the order of system events which must have occurred in the dynamic execution of a program before a join point is reached. While Alpha provides logic predicates to do so (e.g. `before` and `mostRecent`), Tracematches make use of regular expressions over execution traces, and JAsCo provides a notation to define state machines. AspectJ [*Kiczales et al. (2001)*, *Laddad (2003)*] provides the `cflow` pointcut designator which requires that system events must have occurred in "nested" order (i.e. some of them must still be "active", or "on the call stack", when others occur). And the data flow (`dflow`) pointcut designator specifies that some (i.e. the same) data from the current join point context had to be involved in earlier system events, too[24].

All of these language constructs have in common that they explicitly disclose the dependencies that must exist between different system events in order to reach a join point. With all of those selection constraints being explicitly disclosed, it is generally assumed that developers are less prone to overlook important details of the join point selection. Therefore, the language constructs are deemed to facilitate the comprehension task.

4.1.2 Discussion

In a perfect world, there would be one "master" aspect-oriented programming language which incorporates all of the aforementioned language constructs. That language would permit developers to combine these language constructs in arbitrary ways, such that developers can completely specify all characteristics of the situations they want to select concisely and explicitly in just one statement. This would be for the sake of both a swift implementation as well as an easy comprehension. Furthermore, the programming language would come with a compiler for all (or at least for several important) base programming languages (e.g. Java, C++, COBOL, Smalltalk, etc.), so that developers can exploit the join point selection means of the programming language regardless of the base programming language their application is written in.

Among the aspect-oriented programming languages mentioned above, Tracematches and Alpha may be closest to such a "master" aspect-oriented programming language[24]. Unfortunately, though, each of these languages requires its own compiler, and each of these compilers is targeted towards just one particular base programming language (i.e. Java in case of Tracematches, and L2 in the case of Alpha). In consequence, their join point selection means may be exploited by only a limited set of developers (i.e. those whose base application is implemented in Java or L2).

In reality, developers are thus frequently forced to use an aspect-oriented programming language which does not provide (all of) the language constructs that are needed to capture the characteristics of selected runtime situations in just one statement. Examples of such languages are AspectJ, AspectC++ [*Spinczyk et al. (2002)*], AspectCOBOL [*Lämmel & Schutter (2005)*], AspectS [*Hirschfeld (2002)*], Perl Aspect [*Kennedy et al. (2009)*], etc. As a result, developers are frequently forced to implement manual workarounds, such as the ones presented in the previous chapter. Unfortunately, these workaround implementations frequently have bad impacts on the comprehension task (see previous chapter for a discussion). Switching to another aspect-oriented programming language supporting more sophisticated join point selection means may often be no option,

[24] Note that both Alpha and Tracematches provide means to express data flow constraints as well as constraints on the order of system events (which may be required to be "nested"), too.

especially if large parts of the base application may have already been implemented using a particular programming language, e.g., C++, COBOL, Smalltalk, Perl, etc., and their re-implementation using the base language of the respective aspect language, e.g., Java, would be too costly.

In order to overcome this problem, it would be an option for this thesis to complement each aspect-oriented programming language with appropriate language constructs, such that join point selections of dynamic and behavioral join points which resort to information from earlier points in runtime could be implemented in an explicit manner. This would mean to extend the compilers of (or implement a pre-compiler for) AspectJ, AspectC++, AspectCOBOL, AspectS, Perl Aspect, and any other aspect-oriented programming language there is (and there is to come). The problems of this approach are apparent: apart from the fact that this would mean a tremendous amount of work, and that approximately the same work needs to be done for each language over and over again, the approach would not solve the problem once and for all because the problem is likely to reemerge as soon as a new aspect-oriented programming language comes up (which does not provide such explicit means). Therefore, this thesis will pursue a different approach.

4.2 Aspect-Oriented Modeling Approaches with Visual Notations

4.2.1 Overview of Existing Approaches

Apart from the development of aspect-oriented programming languages, a variety of aspect-oriented modeling approaches emerged which aim to express aspects at a higher level of abstractions. Many of these approaches provide visual notations to depict aspects. Again, a major driving force to develop such modeling approaches and their visual notations was to improve the comprehensibility of the software artifacts. That is why this section presents a selection of aspect-oriented modeling approaches and discusses their suitability to address the problem presented in Chapter 3. Note that the discussion in this section is particularly focused on the way in which the selected approaches represent join point selections (despite of the fact that the focus of the approaches may be on something else than "an easy comprehension of complex join point selections").

4.2.1.1 Visual Approaches Using Textual Notations to Represent Join Point Selections

Some aspect-oriented modeling approaches make use of textual notations to specify join point selections. One subgroup thereof is concerned with model merging, which can be used to weave aspect models with base models. Examples are the Theme/UML approach [*Clarke & Baniassad (2005)*] as well as the Role Models approach [*France et al. (2004)*]. In these approaches, selection of join points means identification of model elements which represent the same concepts in the aspect model and the base model, and which thus need to be merged into one model element. To do so, the approaches usually only support join point selection by name. In the Role Models approach, for example, developers are required to concretize abstract aspect models with the names of model elements[25] of the base (or "primary") model, such that the concretized (or "context-specific") aspect models can subsequently be merged with the base model using a name-based composition procedure. This is done by defining name bindings in the form of

[25] These model elements (of the base, or "primary", model) represent the join points.

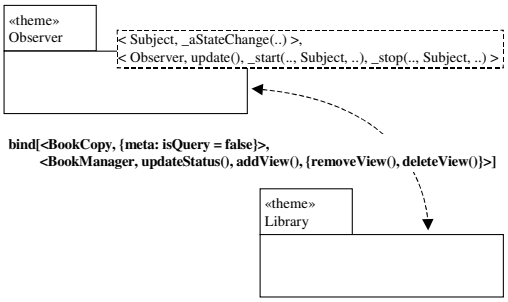


Figure 4.1 Representing join point selections in Theme/UML (cf. [*Clarke & Baniassad (2005)*]).

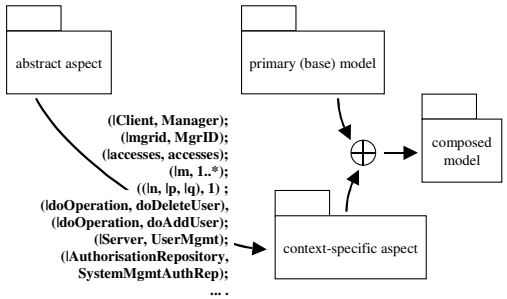


Figure 4.2 Specifying join point selections in the Role Models approach (cf. [*France et al. (2004)*]).

"(|aspect element name, model element name);" (see Figure 4.2 for an illustration[26]). Theme/UML offers a special "bind" relationship to do pretty much the same thing. This time, the annotation tagged to the "bind" relationships specifies the model elements of the base model which are to be bound to the parameters of the aspect models (the list of model elements in the annotation and the list of parameters in the parameter box of the aspect need to be structurally compatible; see Figure 4.1 for an illustration[26]).

As a specialty, Theme/UML also offers means to select join points based on other properties than their names using so-called "meta-queries", which are defined in terms of the Object Constraint Language (OCL) [*Warmer & Kleppe (2003)*]. This feature of Theme/UML may be exploited to constraint properties of the static and structural context in which selected model elements must occur. For example, the meta-query shown in Figure 4.1 selects all model elements whose property "isQuery" is set to "false".

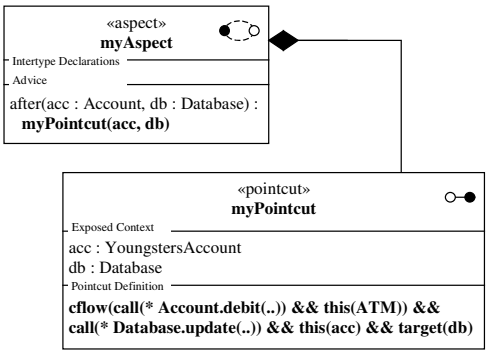


Figure 4.3 Representing join point selections in the Bottom-Up Approach (cf. [*Kandé et al. (2002)*]).

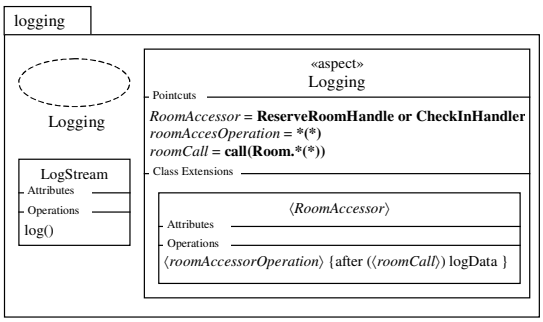


Figure 4.4 Representing join point selections in AOSDw/UC (cf. [*Jacobson & Ng (2004)*]).

Another group of aspect-oriented modeling approaches choose to use or slightly adopt the pointcut languages of existing aspect-oriented programming languages. The Bottom-Up Approach [*Kandé et al. (2002)*] uses the core pointcut language of AspectJ[27], for example, while the Aspect-Oriented Software Development Approach with Use Cases

[26] Note that for brevity, both the Role Models approach as well as the Theme/UML approach permit to map an aspect element to more than one base element at the same time (e.g. using braces { } or parentheses ( )); see [*Clarke & Baniassad (2005)*] and [*France et al. (2004)*] for further explanations.

[27] Another approach which uses the pointcut language of AspectJ in order to express join point selections has been presented by the author of this thesis earlier (cf. [*Stein et al. (2002)*]).

(AOSDw/UC) [*Jacobson & Ng (2004)*] uses a slight variation of it. Illustrations of these approaches are shown in Figure 4.3 and Figure 4.4. The pointcut definition in Figure 4.3 (see bold text) is pure AspectJ. And the components of the pointcut definition in Figure 4.4 (see bold and italic text) refer to particular AspectJ language constructs, each (i.e. there exists a one-to-one mapping).

Another approach adopting an existing pointcut language has been presented by [*Kellens et al. (2006)*]. The approach suggests Model-Based Pointcuts as a means to specify join point selections in terms of conceptual models rather than source code. The underlying assumption of the approach is that models change less frequently than program code, and that Model-Based Pointcuts are thus more robust (and less **fragile** [*Störzer & Graf (2005)*]) against program changes. The approach adopts the pointcut language of CARMA (formerly known as Andrew) [*Gybels & Brichau (2003)*], and extends it with a new predicate called `classifiedAs` which refers to entities[28] in the conceptual model of a program. For example, Figure 4.5 presents a join point selection (see bold text in the note symbol) which refers to all invocations of methods of an "OutputGeneration" entity which are invoked by an instance of a "PageElements" entity.

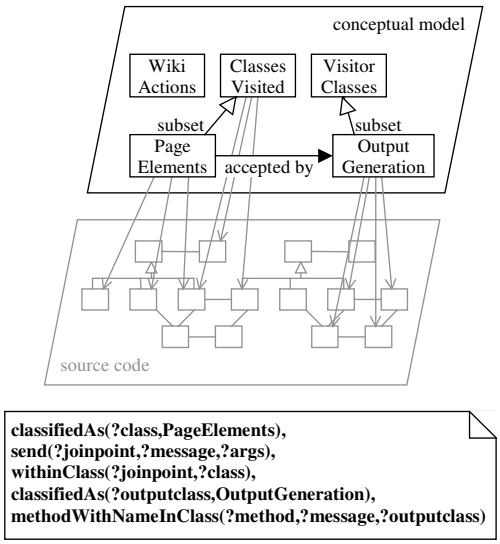


Figure 4.5 Representing join point selections as Model-Based Pointcuts (cf. [*Kellens et al. (2006)*]).

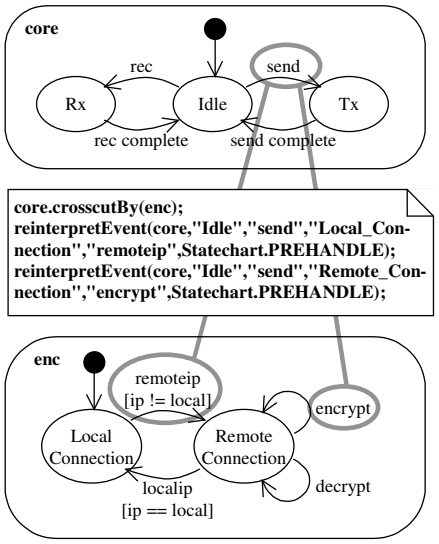


Figure 4.6 Representing join point selections in AOSF (cf. [*Mahoney et al. (2004)*]).

Last but not least, the Aspect-Oriented Statechart Framework (AOSF) [*Mahoney et al. (2004)*] shall be presented as a representative of approaches that introduce aspects to state machines. Similar to the model merging approaches described above, AOSF offers a textual notation to designate corresponding elements (i.e. events, in this case). Unlike the model merging approaches, though, models are not merged. Instead, an execution semantic (called "event reinterpretation") is defined which specifies how the models are supposed to interplay (i.e. be executed) together. The aspectual specification shown in Figure 4.6 illustrates how such "event reinterpretation" may be defined (see bold text in the note symbol).

[28] Entities in the conceptual model are expressed as *intensional views* (cf. [*Mens et al. (2006)*]) on the source code.

4.2.1.2 Visual Approaches Providing Visual Notations to Represent Join Point Selections

Some approaches use behavioral diagrams (rather than textual notations as in the previous cases) to represent join point selection constraints on earlier system behavior. One subgroup of these approaches visualizes both join point selection and join point adaptation in just one diagram. Examples of such approaches are the Superimposition Approach [*Katara & Katz (2003)*] as well as the Modeling-of-Aspects-using-a-Transformation-Approach (MATA) [*Whittle et al. (2009)*]. In order to distinguish join point selection from join point adaptation, the approaches use different colors or special markers.

The Superimposition Approach, for example, paints join point selections in gray-shaded color, while join point adaptations are drawn in solid black (see Figure 4.7). Accordingly, the model shown in Figure 4.7 selects the "AudioController" state chart which comprises a "Record-on", "Idle", and "Alarm-on" state and which has a "stop" transition from the "Record-on" to the "Idle" state. Subsequently, this state chart is adapted as follows: the existing "Record-on" state is complemented with two substates ("Normal" and "Alarm-in-Record-on"); the existing "stop" transition to state "Idle" is augmented with a new guard (i.e. "[IN Normal]"); and a new "stop" transition from state "Record-In" to state "Alarm-in" is added which bears the guard "[IN Alarm-in-Record-on]".

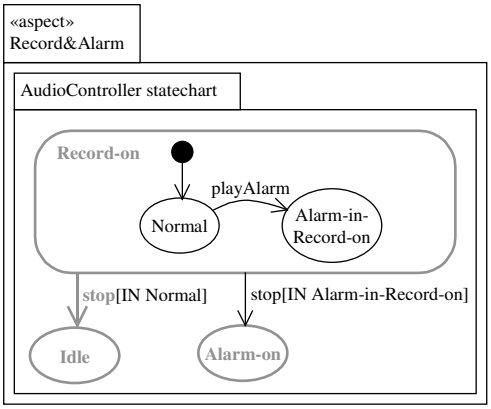


Figure 4.7 Representing join point selections in the Superimposition Approach (cf. [*Katara & Katz (2003)*].)

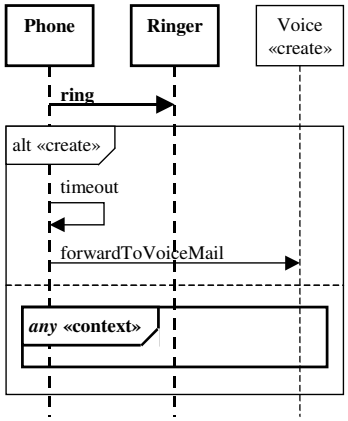


Figure 4.8 Representing join point selections in MATA (cf. [*Whittle et al. (2009)*]).

MATA marks join point adaptations with a special stereotype, i.e. «create» or «delete», while join point selections are generally left unmarked (a special stereotype «context» may be used to mark join point selection elements when they are contained in join point adaptation elements). For example, the model shown in Figure 4.8 selects all invocations of method "ring" issued by an instance of "Phone" and addressed to an instance of "Ringer", and inserts (i.e. "creates") a new "alt" fragment after it. That "alt" fragment contains an "any" fragment in its second region which refers to all messages that come after the selected "ring" call (i.e. its "context"). This means that the newly created "alt" fragment reuses (or wraps around) all of such messages in its second region.

Another group of aspect-oriented modeling notations represents join point selection and join point adaptation in distinct (but related) diagrams. Examples are the Motorola WEAVR [*Cottenier et al. (2007b)*], Reusable Aspect Models (RAM) [*Kienzle et al. (2009)*], and High-Level Aspects (HiLA) [*Zhang et al. (2007)*].

The Motorola WEAVR provides a notation to select transitions[29] and/or actions from Specification and Description Language (SDL) models. Figure 4.9 shows an example. The "pointcut" shown on the left side of Figure 4.9 intercepts all transitions leaving from state "Init" and leading to any arbitrary state (" '*' ") which are triggered by action "access()". The selected transitions (i.e. all actions that are executed during those transitions) are replaced by the "connector" shown on the right side of Figure 4.9 (where "proceed();" leads to the execution of the initially intercepted (sequence of) actions).

The RAM approach provides similar notations as the Motorola WEAVR, yet for UML state charts and UML sequence diagrams. No illustration is shown here. However, the approach is based on a semantic-based formalism to weave Message Sequence Charts (MSC) [*ITU (1999)*] (which has been presented in [*Klein et al. (2006)*] and has been extended in [*Klein et al. (2007)*]). That approach is illustrated in Figure 4.10. The aspect shown in Figure 4.10 selects message sequences consisting of a message "new attempt" which is immediately followed by another message "try again". The aspect complements that message sequence with an extra message "save attempt" which is introduced after message "new attempt" and before message "try again".

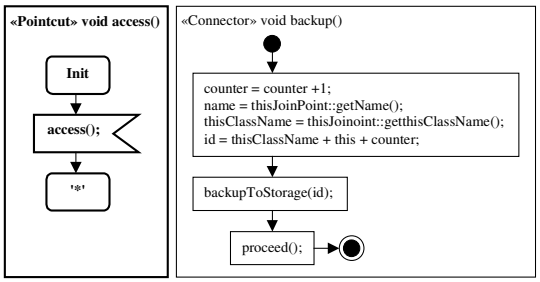


Figure 4.9 Representing join point selections in the Motorola WEAVR[30] (cf. [*Cottenier et al. (2007b)*]).

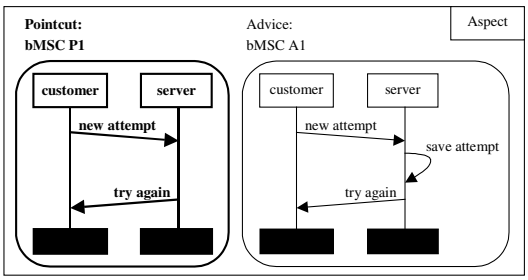


Figure 4.10 Representing join point selections in Semantic-based Weaving of Scenarios (cf. [*Klein et al. (2006)*]).

HiLA [*Zhang et al. (2007)*] is another approach which visually separates join point selection from join point adaptation. HiLA extends UML state charts to represent aspects (see Figure 4.11). It permits to specify transition pointcuts and configuration pointcuts. A transition pointcut may intercept all transitions from a particular state configuration to another particular state configuration. A configuration pointcut may intercept all transitions from or to a particular state configuration, irrespective of any subsequent or previous state configuration (respectively). For example, Figure 4.11 depicts a transition pointcut which selects all transitions from a state configuration containing state "SelectLevel" to another state configuration containing state "ShowQuestion" where the value of the local variable "i" is greater than zero (the label "«before»" indicates the point in time – relative to the transition – at which the advice is to be executed).

Furthermore, HiLA permits to specify selection constraints on earlier system states with help of so-called trace variables. To do so, trace variables specify state configurations which must occur in the earlier execution history of a program; in addition to that, they

[29] The notion of a "transition" is very specific in WEAVR: it refers to all (sequences of) actions that will be executed when the system changes from the start state (e.g. state "Init in Figure 4.9) to the end state (e.g. state "'*'" in Figure 4.9); see [*Cottenier et al. (2007a)*] to find out how these (sequences of) actions are inferred from the SDL models.

[30] The advice-pointcut binding specification is omitted in Figure 4.9 for reasons of brevity; see Figure 4.16 for an example of a complete aspect specification.

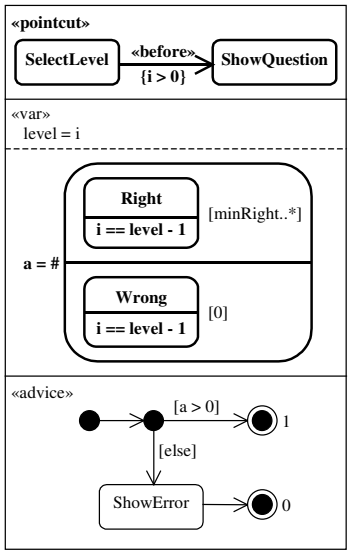


Figure 4.11 Representing join point selections in HiLA (cf. [*Zhang et al. (2007)*]).

determine *how often* these state configurations must occur in the execution history. For example, trace variable "a" in the aspect shown in Figure 4.11 specifies two state configuration, and determines that one of these state configurations must occur at least minRight times (i.e. "[minRight..*]"), while the other state configuration must not occur at all (i.e. "[0]"). The value of the trace variable "a" indicates how many sequences of state configurations in the earlier execution history of the program have satisfied these constraints. It is evaluated in the advice in order to determine the ultimate aspectual behavior.

Further approaches that provide distinct representations for join point selections and join point adaptations include Larissa [*Altisen et al. (2006)*] and A-LTS [*Yagi et al. (2007)*]. In Larissa and A-LTS, join point selections are represented with help of automata.

Larissa permits to specify pointcuts in terms of Mealy automata which synchronously observe all inputs and outputs of the original program, and emit a new output each time a join point is reached. The pointcut automaton shown in Figure 4.12, for example, switches states upon every occurrence of an (output) command "PastOn" or "PastOff", and it indicates the occurrence of a join point whenever it is in state "X" and observes an (input or output) command other than "PastOff". It does so by emitting a new (output) command "JP".

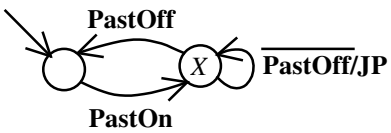


Figure 4.12 Representing join point selections in Larissa [*Altisen et al. (2006)*]

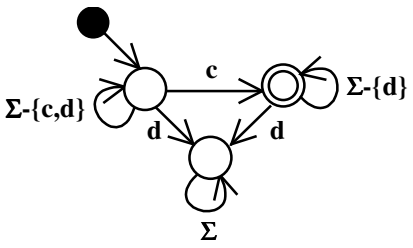


Figure 4.13 Representing join point selections in A-LTS (cf. [*Yagi et al. (2007)*]).

Similar to Larissa, A-LTS represents pointcuts with help of finite deterministic automata. These automata monitor the event sequence performed by the base program. A join point is found whenever the pointcut automaton switches to a final (i.e. accepting) state. For example, the pointcut automaton shown in Figure 4.13 accepts all event sequences containing at least one "c" but no "d". In consequence, all events occurring after the first "c" and before the first (subsequent) "d" are selected as join points.

4.2.2 Discussion

Despite the variety of aspect-oriented modeling approaches that have emerged so far, none of these approaches is sufficient to overcome the problems that have been addressed in Chapter 3. The following discussion illustrates with help of examples why the presented approaches are insufficient. The sanitizing aspect (see section 3.1.1) is chosen as a running example.

4.2.2.1 Model-Merging Approaches

The first group of aspect-oriented modeling approaches discussed here are model merging approaches like Theme/UML and Role Models. The problem of such approaches is that they usually only support the selection of join points based on static and structural properties, such as their names. As a result, developers are required to implement complex workarounds as soon as they need to deal with join point selections that constrain earlier system behavior. These workarounds impose a significant burden on developers who need to understand the behavior of an aspect.

Figure 4.14 gives an illustration. It shows a Theme/UML model of the sanitizing aspect which has been discussed in section 3.1.1. The model defines two data stores (i.e. two collections named "cookies" and "values") as well as three advice (i.e. three sequence diagrams named "getCookie", "getValue", and "print") in order to perform the selection of the right set of join points.

In order to understand the Theme/UML model in Figure 4.14, readers need to acquire (at least the very basics about) the notation and the semantics of the Theme/UML approach, first. In particular, they need to know about the Theme/UML way to specify `around` advice (which are the only kind of advice supported by Theme/UML): `around` advice are specified with help of sequence diagrams, and they overwrite the original behavior of the bound methods (i.e. the sequence diagram "getValue" redefines the behavior of method "getValue" of class "Cookie" in Figure 4.14). Furthermore, readers need to know that the bound methods themselves are renamed, i.e. their names are prefixed with "_do_" (see method "_do_getValue" in class "Cookie" in Figure 4.14, for example). This is necessary so that the original behavior of the bound methods may still be invoked by calling the altered method names (likewise to calling `proceed` in AspectJ; see the invocation of method "_do_getValue" in sequence diagram "getValue" in Figure 4.14).

Once the readers have acquired all that knowledge about the approach, they need to look very carefully for the interdependencies between the different advice, which result from the shared use of common data stores. In case of the "Sanitizing" theme shown in Figure 4.14, this means to find out how each of the advice accesses the global collections "cookies" and "values", and to recognize how these data accesses affect the order in which the two advice need be executed.

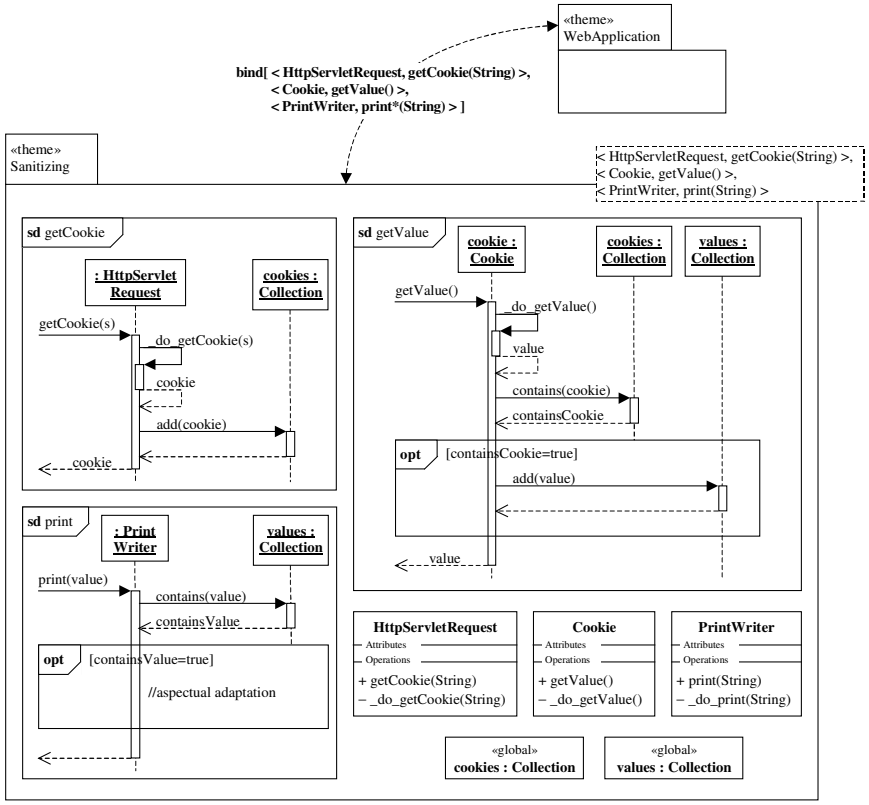


Figure 4.14 Representing the Sanitizing Aspect with Theme/UML[31].

In summary, readers suffer from very similar problems (like the ones discussed in Chapter 3) when they need to deal with a complex implementation of a join point selection in the mentioned modeling approaches. This is because they need to investigate a multitude of pointcut and advice specifications before they are able to estimate the proper effects of the aspect. Only one of these pointcut and advice specifications (e.g. sequence diagram "print" in Figure 4.14) is actually concerned with performing the aspectual adaptation, while the remaining pointcut and advice specifications (e.g. sequence diagrams "getCookie" and "getValue" in Figure 4.14) are only needed to identify the right set of join points.

4.2.2.2 Adoption of Existing Pointcut Languages

The situation is not much different in case of modeling approaches which adopt the pointcut languages of existing programming languages, e.g. in the Bottom-Up Approach or in the AOSDw/UC approach. Such approaches are usually capable of expressing at least *some* selection constraints on the earlier system behavior. For example, the Bottom-Up Approach permits to specify selection constraints on the control flow in which join points must occur (using the `cflow` pointcut designator of AspectJ). And the AOSDw/UC approach permits to require that a particular join point needs to come to pass in the immediate execution context of a particular method (this is a dynamic interpretation of the selection semantics of the `withincode` pointcut designator of AspectJ). However, since the join point selection means of those approaches are so closely related to the pointcut

[31] Note that class parameters and method parameters may given arbitrary names; in the example presented here, though, parameters have the same name as the classes and methods they are bound to in order to facilitate the comprehension of the theme.

languages they are adopting, they also suffer from the same deficiencies as those pointcut languages:

For example, Figure 4.15 illustrates an implementation of the sanitizing aspect (see section 3.1.1) using the Bottom-Up Approach. The figure reveals the close similarity of the aspect implementation in the Bottom-Up Approach with the aspect implementation in AspectJ (which has been presented in section 3.1.1). Similar to the AspectJ aspect, the aspect shown in Figure 4.15 defines three pairs of pointcut and advice as well as two collections, which are maintained and evaluated by the three pairs of pointcut and advice in order to select the right set of join points.

And like in the case of AspectJ, readers of the aspect have to carefully investigate all three pairs of pointcut and advice before they are able to understand what join points are ultimately affected by the aspect. They need to zoom into the advice bodies (which are defined in another (behavioral) diagram and are not shown in Figure 4.15) in order to understand how these pointcuts and advice interdepend on each other, which means that they need to carefully examine their shared access to the common data structures.

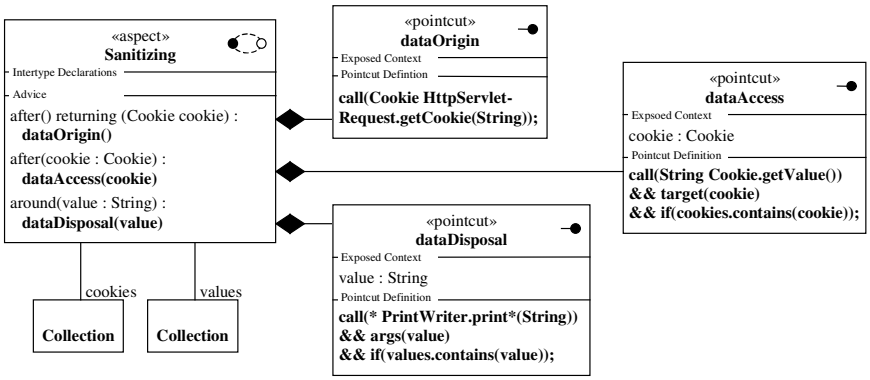


Figure 4.15 Representing the Sanitizing Aspect in the Bottom-Up Approach.

The burdens of reading and understanding complex join point selections defined with modeling approaches adopting an existing pointcut language (i.e. the Bottom-Up Approach, in this case) are thus pretty much the same as with the aspect-oriented programming language that has defined that pointcut language (i.e. AspectJ, in this case).

4.2.2.3 Incapability to Express Chronological Dependencies

Among the modeling approaches which represent join point selections with help of behavioral models, there is no approach which is capable to overcome the problems outlined in Chapter 3 either. Some of the approaches merely permit to specify join point selections on singular system events (e.g. on singular transitions or actions, such as the Motorola WEAVR or the AOSF approach), or on consecutive system events (e.g. on so-called "strict" sequences of messages, such as defined and supported by [*Klein et al. (2006)*]). Other approaches are incapable to determine the order in which system events must occur (e.g. HiLA only permits to constrain the frequency with which a particular state configuration must occur in the execution history of a program; yet, it may not impose a partial order on those state configurations). In consequence, it is not possible to specify a join point selection on system events that have to occur in some (partial) order, yet not necessarily one after the other. This is a severe limitation of the expressiveness of the

approaches. For example, it is not possible to specify a complex join point selection like the ones presented in sections 3.1.1 and 3.1.2 (see Chapter 3) in just one pointcut diagram – unless the selected system events are implemented to occur right after each other in the affected base models (which is highly unlikely to be the case in the scenarios outlined in Chapter 3).

Hence, in order to specify join point selections like the ones presented in sections 3.1.1 and 3.1.2, developers need to manually keep track of all relevant system events that come to pass during the execution of the program – so that they can decide at the (final) join point if the adaptation needs to be performed or not. To do so, the developers have to select and adapt all system events which may play a role in that (non-consecutive) sequence of system events. An illustration is given in Figure 4.16 which displays an implementation of the sanitizing aspect which has been discussed in section 3.1.1 in the Motorola WEAVR. The aspect defines three pairs of pointcut and advice, each of which selects and adapts a different (relevant) system event and accesses the common data store (i.e. collection "cookies", "values", or both), either to store data that is needed by another advice, or to evaluate if the core behavior of the advice needs to be executed.

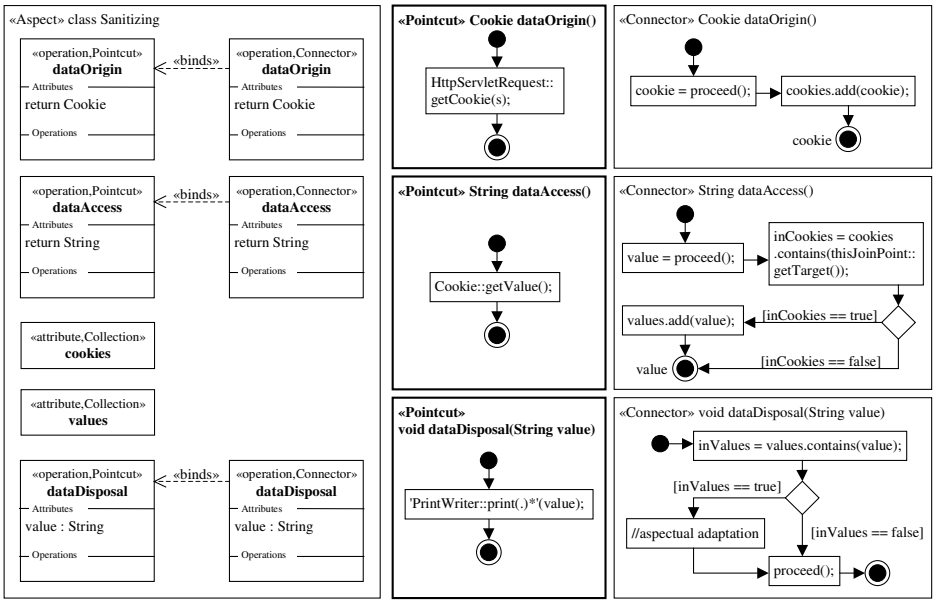


Figure 4.16 Representing the Sanitizing Aspect in the Motorola WEAVR.

In the end, readers are once again confronted with a new notation to specify aspects which they need to acquire before they are able to properly understand the aspect specification. Furthermore, they are once again obliged to inspect a conglomeration of interdependent join point selections and join point adaptations in order to find out what runtime situations are finally affected by the aspectual adaptation.

4.2.2.4 Static Join Point Model

Other approaches do permit the specification of join point selections on non-consecutive system events. However, the selection capabilities of these approaches are frequently impaired by the necessity that the non-consecutive system events must be found in *one* consecutively specified sequence of system events in the base model. An example of such an approach is MATA.

Cause of this problem is that the approaches rely on a purely static join point model. This means that their join point selections designate a spatial location in an existing model, which must be found by some pattern-matching algorithm before it is going to be modified according to the join point adaptation. It is one of the inherent characteristics of these approaches that they fail to find matches (i.e. join points) as soon as a join point selection designates elements (i.e. concepts, such as method calls, state transitions, actions, etc.) that are defined or located in different parts of the model, or in different submodels [*Klein et al. (2006)*, *Grønmo et al. (2008)*][32].

An illustrative example of such a situation is given in [*Klein et al. (2006)*]. The example presents a scenario in which the selection of two consecutive system events fails. The example makes use of Highlevel Message Sequence Charts (HMSC) [*ITU (1999)*], which determine in which order basic Message Sequence Charts (bMSC) are executed. The example is about applying the aspect model shown in Figure 4.17 (left part; which is identical to Figure 4.10) to the base model shown in Figure 4.17 (right part). The aspect model aims to adapt all messages named "new attempt" and "try again" which occur in consecutive order, and where the first message ("new attempt") is sent from a "customer" instance to a "server" instance and the second message ("try again") is sent from the (same) "server" instance to the (same) "customer" instance. Upon each occurrence of such a message pattern, the advice is supposed to perform some extra behavior ("save attempt") in between the selected messages.

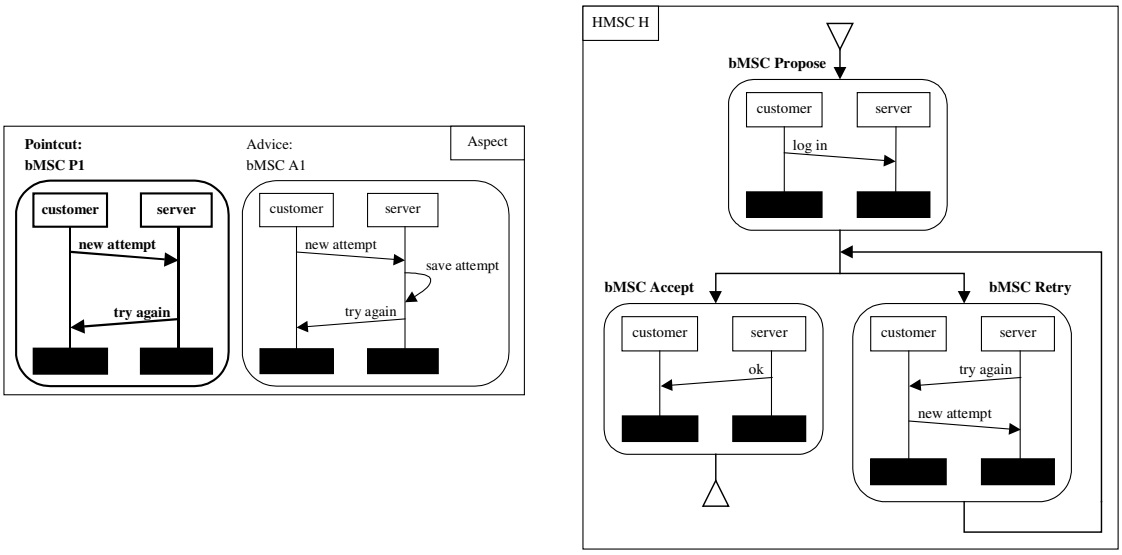


Figure 4.17 A sample aspect, and a sample base model in HMSC (cf. [*Klein et al. (2006)*]).

As it can be easily seen from the figures, a purely syntactic pattern-matching of the pointcut of the aspect model with (arbitrary) parts of the base model will not yield any results. At the same time, however, it is clear (from the semantics of HMSC) that the message sequence pattern outlined in the pointcut may nonetheless occur in the dynamic execution of the HMSC (e.g. when the loop containing the bMSC "Retry" executes more then once) – and thus the aspectual adaptation should apply. In such situations, approaches

[32] Note that basically all model merging approaches are afflicted to this problem, too, since they generally rely on a purely static join point model, as well.

like MATA which rely on a purely static join point model generally fail to select all relevant message sequence patterns.

In order to overcome this insufficiency (i.e. in order to have aspectual adaptations take effect at all appropriate places), developers are generally required to refactor their join point selections. They need to break them apart into a collection of join point selections and join point adaptations, each addressing one of the system events of interest. For example, Figure 4.18 illustrates what a refactoring of the join point selection shown in Figure 4.17 could look like. The aspect definition in Figure 4.18 defines two pairs of pointcut and advice. The first one sets a flag after a customer has sent a new attempt, thus putting the system into state "new attempt". The second one evaluates that system state whenever the server asks for a new try, and saves the attempt (only) if the system is in state "new attempt".

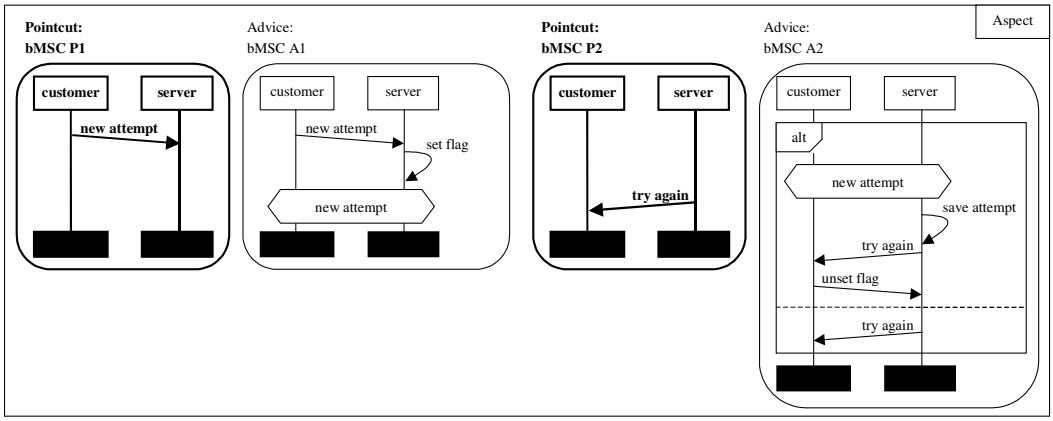


Figure 4.18 Refactoring of the join point selection shown in Figure 4.17.

In the end, readers of the join point selection are faced with the same burdens that have been discussed in Chapter 3: first, they need to be able to read the diagrams (e.g. they need to know how system states may be represented in MSC), and then they need to analyze the interdependencies between the join point selections and join point adaptations in order to recognize which systems events are eventually affected by the aspect.

4.2.2.5 Incapability to Express Data Dependencies

Note that there are interesting approaches which try to overcome the aforementioned problem at least to a certain extent. For example, [*Klein et al. (2006)*] present a semantic-based weaving approach which weaves bMSC based on the execution semantics of HMSC (rather than on pure syntactic pattern matching). Similar to that, [*Klein et al. (2007)*] and [*Grønmo et al. (2008)*] present approaches which do so for UML sequence diagrams. As a result, these approaches free developers from refactoring join point selections like the one shown in Figure 4.17. Nevertheless, the approaches suffer from significant limitations. In particular, they do not (permit to) consider data dependencies on message parameters that must be fulfilled by the messages, nor do they permit to constrain "nesting" of the messages (such as it may be done with the `cflow` constraint in AspectJ). In consequence, the approaches are not able to cope with all of the complex join point selections that have been presented in Chapter 3. In particular in case of the scenarios presented in sections 3.1.1 and 3.1.2, the approaches remain afflicted with the same substantial problems as they have been outlined above.

Other approaches that perform weaving based on execution semantics rather than on syntactic pattern matching are Larissa and A-LTS. Larissa and A-LTS define join point selections in terms of automata. Likewise to the previously mentioned approaches, neither of them is capable of expressing data dependencies that must be fulfilled by (a sequence of) system events because neither of them considers method arguments that may be passed along with the system events triggering a state transition. Consequently, Larissa and A-LTS are incapable of dealing with join point selections such as the ones discussed in sections 3.1.1 and 3.1.2 of Chapter 3, too.

Due to the incapability of the approaches to express data dependencies between (arguments and return values of) relevant system events, it is not possible to exemplify the sanitizing aspect with help of one of the approaches.

4.3 Summary

In summary, it remains to attest that the capabilities of existing aspect-oriented software development approaches to facilitate the comprehension of complex join point selections (of dynamic and behavioral join points) are insufficient.

At the programming level, various language constructs have been proposed so far which promise to facilitate both the specification as well as the comprehension of a complex join point selection. However, each of these language constructs requires a particular compiler, and these compilers focus on particular programming languages only. In consequence, the language constructs are available to a limited group of software developers only, while the majority of software developers are still forced to implement their complex join point selections manually, i.e. with help of multiple and interdependent pairs of join point selections and join point adaptations. The negative impact of this fragmentation of the join point selection on the comprehension task has been discussed and illustrated in Chapter 3.

At the modeling level, multiple modeling approaches have been presented, each of which coming with a particular way to designate join points. However, none of these approaches is capable of expressing all selection constraints that are necessary to specify complex join point selections like the ones discussed in Chapter 3 in just one pointcut definition (Table 4.1 gives a concluding overview of the deficiencies of the modeling approaches discussed in this chapter). One reason is that the approaches are incapable of specifying join point selections on non-consecutive system events (e.g. they only permit to specify join point selections on singular system events, or on consecutive system events). Another reason is that many approaches rely on a purely static join point model, and are thus inherently incapable of defining join point selections on non-consecutive system events which catch all relevant sequences of system events in the base program. Finally, approaches which do provide means to specify join point selections on non-consecutive system events are frequently incapable of expressing all of the chronological dependencies and data dependencies that must exist between these non-consecutive system events. As a result, developers are frequently forced to realize their complex join point selections with help of a multitude of trivial – yet strongly interdependent – join point selections and join point adaptations, just as they need to do it with current aspect-oriented programming languages. And in the end, they are faced with the same comprehension problems as discussed in Chapter 3 when they need to understand a complex join point selection.

Table 4.1 Deficiencies of aspect-oriented modeling approaches.

Incapability to specify selection constraints...				
...on the chronological order of non-consecutive system events				
...on the level of the call stack (such as `cflow` in AspectJ)				
...on the involvement of same data in different system events				
Static join point model				
Theme/UML [*Clarke & Baniassad (2005)*]	●	●[1]	●[1]	●[1]
Role Models [*France et al. (2004)*]	●	●	●	●
Bottom-Up Approach [*Kandé et al. (2002)*]		●		●[2]
AOSDw/UC [*Jacobson & Ng (2004)*]	●[3]	●	●[4]	●[4]
Model-Based Pointcuts [*Kellens et al. (2006)*]	●	●	●	●
AOSF [*Mahoney et al. (2004)*]		●	●	●
MATA [*Whittle et al. (2009)*]	●			
Superimposition Approach [*Katara & Katz (2003)*]	●			
Motorola WEAVR [*Cottenier et al. (2007b)*]		●		●[5]
Semantic-based Weaving Approach by [*Klein et al. (2006)*]	○[6]	●	●[7]	○[8]
Semantic-based Weaving Approach by [*Grønmo et al. (2008)*]	○[6]	●	●[7]	
RAM [*Kienzle et al. (2009)*]	○[6]	●	●[7]	○[8,9]
HiLA [*Zhang et al. (2007)*]		●	●[10]	
Larissa [*Altisen et al. (2006)*]		●	●	
A-LTS [*Yagi et al. (2007)*]		●	●	

[1] By means of meta-queries, Theme/UML permits to specify similar selection constraints like MATA with help of complex OCL expressions. [2] The Bottom-Up Approach and AODM permit to require that a method call must occur in the control flow of a particular other method (using AspectJ's `cflow` constraint). [3] join points in AOSDw/UC can be dynamically interpreted, yet always can be statically computed. [4] AOSDw/UC only permits to require that a method call must occur in the immediate execution context of a particular other method (similar to AspectJ's `withincode` constraint). [5] the Motorola WEAVR supports the selection of actions that have been triggered by a transition (i.e. that come to pass in the control flow of that transition). [6] the semantic-based weaving approaches by [*Klein et al. (2006)*], [*Grønmo et al. (2008)*], and RAM rely on static model analysis (i.e. flow analysis); support of selecting dynamic join points is thus limited. [7] the semantic-based weaving approaches by [*Klein et al. (2006)*] and [*Grønmo et al. (2008)*] and RAM only permit to require that a method call must occur in the immediate execution context of a particular other method (similar to AspectJ's `withincode` constraint). [8] the semantic-based weaving approaches by [*Klein et al. (2006)*] and RAM may be "configured" to support the selection of non-consecutive system events in sequence diagrams. [9] support for specifying selection constraints on the chronological order of non-consecutive system events with help of UML state charts is planned in RAM, but not supported yet. [10] specifying selection constraints on the level of the call stack with help of state charts requires supplementary means to map corresponding invocation and termination events.

Chapter 5

Join Point Designation Diagrams

This chapter introduces Join Point Designation Diagrams (or JPDDs, in short) as means to improve the comprehension task of software developers when they need to understand complex join point selections.

To do so, this chapter first gives a general overview over the basic concepts and ideas which JPDDs are based on (section 5.1). Afterwards, an informal introduction to JPDDs is given which introduces the general structure of JPDDs (section 5.2) and explains its general semantics (sections 5.3 and 5.4). Subsequently, the section outlines what join point properties may be constrained by JPDDs (section 5.5), and introduces the notational means that may be used in JPDDs to define such constraints (sections 5.6, 5.7, and 5.8). In section 5.9, the usage of the notational means is illustrated with help of examples. Section 5.10 concludes this section with a summary.

5.1 Overview

JPDDs are a visual means to represent aspect-oriented join point selections. JPDDs describe all characteristics that selected join points must posses. Or, in other words, they outline the selection criteria that selected join points must satisfy.

JPDDs are based on the Unified Modeling Language (UML) [*Booch et al. (1998)*], meaning that they use and extend the notation of the UML (i.e. its concrete syntax) as well as its conceptual framework (i.e. its metamodel). The notational means of the UML are used (and extended) to express selection criteria on the various facets of a piece of software, such as its structure, its control flow, its data flow, its state transitions (see Figure 5.1). By resorting to the full notational power of the UML, JPDDs are capable of reflecting on the particular conceptual views on program execution which a join point selection may be referring to (e.g. a control flow-, data flow-, or state-oriented join point selection; cf. [*Stein et al. (2006)*]). How this is done is explained in this chapter.

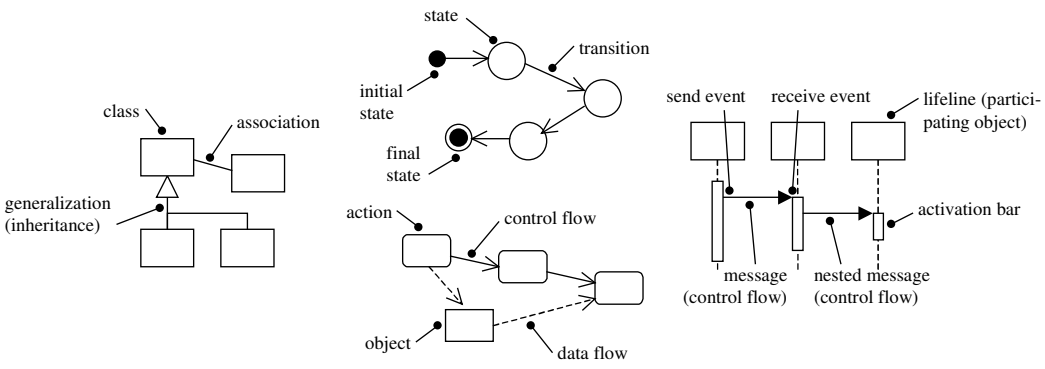


Figure 5.1 Notational means of the UML which are adopted by JPDDs.

The conceptual framework of the UML is used to denote the kinds of elements which are/can be selected by JPDDs. Accordingly, JPDDs can be used to select elements from the specification of a piece of software, e.g. a class or a method declaration (the UML specification refers to this as the "type level"; cf. [*Booch et al. (1998)*]). Furthermore, they can be used to select elements from the runtime context of a software, e.g. a runtime occurrence and a runtime instance (the UML specification refers to this as the "instance level"; cf. [*Booch et al. (1998)*]). The selection of the latter is the primary focus of this thesis.

The way in which selections may be specified with help of JPDDs is inspired by the idea of Query-By-Example [*Zloof (1977)*]. Query-By-Example (QBE) is a common technique in the database domain, and is realized by the standardized Structured Query Language (SQL). The idea is to specify a sample entity with sample properties, and determine how selected entities must compare to that sample entity. "Operators" are used to require equivalence or non-equivalence with the sample properties of the sample entity (e.g. "=" or "<>" in SQL), or to permit a certain degree of deviation form these properties (e.g. "<", ">", "LIKE", "BETWEEN", etc. in SQL). JPDDs, too, describe a (collection of) sample entities with sample properties, and determine how selected entities must compare to those sample entities. JPDDs come with a variety of means to specify deviations (such as wildcards, regular expressions, path expressions, etc.). These deviation means are introduced in this chapter. Another important feature that JPDDs share with the idea of Query-By-Example is that queries are specified in a declarative way, meaning that they specify what is to be retrieved rather than how it is to be retrieved.

5.2 General Structure

JPDDs generally consist of a dashed rectangle with rounded edges. JPDDs have a name, which is located at their upper left corner. At their lower right corner, JPDDs have a so-called **output parameter box**. The output parameter box indicates which elements of the JPDD are going to be part of the selection result, i.e. it designates all elements which are exposed to the join point adaptation and may be modified by it. To do so, the output parameter box lists the **identifiers** of those elements. Identifiers are variable names, which may be prepended to any named element of the JPDD. Visually, identifiers are enclosed in angle brackets, and are prepended by a question mark (e.g. "<?id>"). All elements inside the rounded rectangle of a JPDD (except identifiers) represent selection constraints.

The visual appearance of JPDDs is inspired by UML templates, which are rendered with a "template parameter box" at their upper right corner. This is because both JPDDs and UML templates are considered to denote some kind of "pattern". However, while UML templates represent a generation pattern which may be used to produce new (model) elements, JPDDs denote a selection pattern which is used to retrieve existing elements from a (running) program (how this is done is subject of the next subsection). In contrast to the parameter box of a UML template, the parameter box of JPDDs is located at their lower right corner in order to emphasize this difference and in order to avoid any confusion.

Figure 5.2 shows an example. The JPDD presented here is named "SelectingAMethodCall". The JPDD is outlined by a dashed rectangle with rounded edges. It contains two lifeline symbols and a message symbol. The message symbol is given an

identifier "<?jp>", which is placed in front of the method name "m1". Likewise, both lifeline symbols are given identifiers too, "<?s>" and "<?r>", which are placed in front of the all-quantifiers ".*" (all-quantifiers may be used to select elements regardless of their names; cf. section 5.6). The identifiers are included in the output parameter box of the JPDD at its lower right corner, denoting that the JPDD selects and returns (combinations of) such elements which thus may be used and/or modified by the join point adaptation.

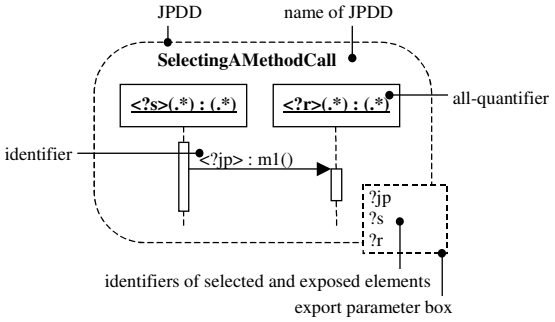


Figure 5.2 General structure of a JPDD.

5.3 General Semantics

The JPDDs considered in this thesis select method calls from the dynamic execution of a program, together with the objects that are involved in such method calls[33]. In order to do so, JPDDs define selection constraints which are expressed by the symbols inside the rounded rectangle of a JPDD, and which must be satisfied by every (combination of) elements that is going to be part of the selection result[34].

As such, JPDDs comply to the idea of "pattern matching" as described by [*Gybels & Brichau (2003)*] in the sense that JPDDs "describe what is common to all the join points that should match the pointcut" [*Gybels & Brichau (2003)*]. Apart from that, they look at program execution as a sequence of events, as it is done by [*Douence et al. (2001)*, *Walker & Viggers (2004)*, *Allan et al. (2005)*], too. And similar to them, they render a (regular) pattern over that sequence of events.

Note that – similar to Tracematches [*Allan et al. (2005)*] – JPDDs may yield multiple parameter bindings for a successful join point match. JPDDs make no assumptions how to cope with these during the join point adaptation[35]. Likewise, they make no assumptions whether the join point adaptation is executed before, around, or after the successful join point match. Such join point adaptation issues are beyond the scope of JPDDs.

Figure 5.3 shows an example JPDD named "SelectingAMethodCall" which selects and returns combinations of one method call and two objects. The selection constraints of the JPDD are specified with help of a message symbol and two lifeline symbols. The message symbol represents a method invocation of a method named "m1" which takes (exactly) one

[33] JPDDs may also be used to select other kinds of program elements, such as class declarations or method declarations, etc. (see [*Stein et al. (2004)*] for further explanations).

[34] This compares to logic unification [*Clocksin & Mellish (2003)*] of the individual elements of the JPDD based on predicates that reflect the elements' properties and relationships as defined by the UML metamodel.

[35] Possible solutions, and their implications, are discussed and presented in [*Allan et al. (2005)*, *Al-Mansari & Hanenberg (2006)*].

"Integer" argument. The lifeline symbols represent two objects, one of which is defined to be an instance of class "Sender", while the other is defined to be an instance of class "Receiver". According to the aforementioned selection semantics, the JPDD shown in Figure 5.3 selects all method calls which invoke a method with name "m1" and which have an instance of class "Integer" as (the only) parameter. Together with each method call, the JPDD furthermore returns the sender and receiver objects of the method call, which are required to be instances of class "Sender" and class "Receiver", respectively.

Figure 5.3 shows a matching method call on the right side. The method call is selected because its properties comply to all selection constraints that are defined by the JPDD: the method call invokes a method named "m1" and passes on an argument which is an instance of class "Integer"; furthermore, the sender object of the method call is an instance of class "Sender", and the receiver object of the method call is an instance of class "Receiver".

Figure 5.3 also shows a non-matching method call on the right side. This method call is not selected by the JPDD because it conflicts with the constraints made by the JPDD in several respects: first of all, the called method has a different name (i.e. "m2" rather than "m1"); furthermore, the argument of the method call is an instance of another class (i.e. "Float" rather than "Integer"); finally, both the sender and receiver objects are instances of other classes, too (i.e. "S" rather than "Sender", and "R" rather than "Receiver").

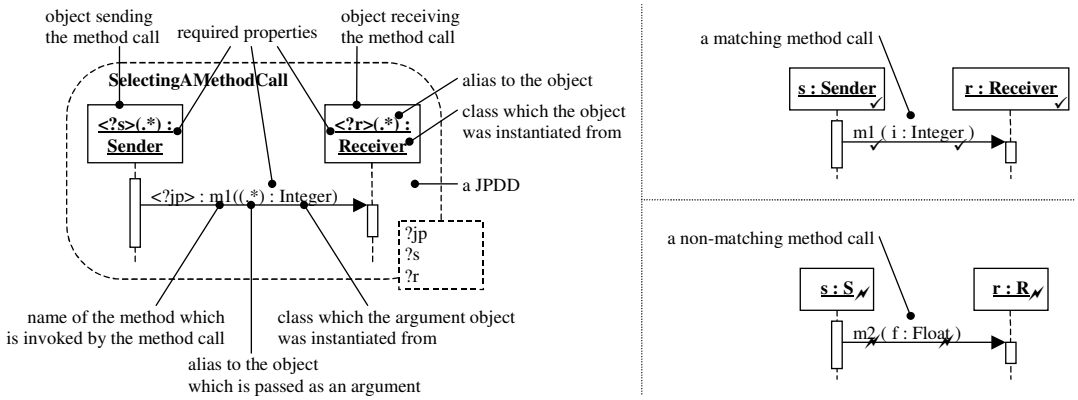


Figure 5.3 Selection semantics of JPDDs by example.

5.4 Combination Relationships

By default, the selection criteria specified in a JPDD are joined together by a Boolean AND. This means that all selection criteria must be fulfilled by the selected elements. Within a JPDD, it is not possible to negate a selection criteria using a Boolean NOT (e.g. to specify that something about an element must *not* be the case) or to specify alternative selection criteria using a Boolean OR (e.g. to specify that just one of two characteristics needs to be satisfied for the element to be selected). If the specification of negations or alternatives is necessary, **combination relationships** (e.g. union or exclusion) must be used to reproduce the respective selection semantics[36]. A third kind of combination

[36] Note that the OCL constraints `{not}` and `{or}` have been used within JPDDs in former publications; their semantics can be realized with help of combination rules, too (as described here), and that is why they are discarded in this thesis for reasons of simplicity.

relationship (i.e. confinement) may be used to complement the selection criteria of one JPDD by the selection criteria of another JPDD (using a Boolean AND).

Combination relationships are introduced to JPDDs in response to the capabilities of many aspect-oriented programming language (e.g. AspectJ [*Laddad (2003)*], AspectC++ [*Spinczyk et al. (2002)*], Alpha [*Ostermann et al. (2005)*], etc.) which permit to specify new join point selections based on existing ones. With help of these means, it is possible to refine or specialize join point selections from more general and more expansive ones, as well as to divide an accumulation of join point selection constraints into smaller – and easier-to-comprehend – subgroups of selection constraints which logically belong together and thus would constitute a meaningful join point selection on their own.

Combination of JPDDs is performed by logically unifying [*Clocksin & Mellish (2003)*] the identified elements of the including/source JPDD with the exposed elements of the included/target JPDD of the combination relationship. Unification may only be performed on elements of the same type (i.e. two elements being unified must both be two classes, two objects, two message calls, etc.). The only exception thereof is the possibility to unify a (static) method call statement and a (dynamic) method call (an example is given in the next section – see Figure 5.5 – which also addresses the difference between static and dynamic elements). This exception has been made because the UML does not allow to differentiate between the two visually. In case of that exception, the (dynamic) method call is implicitly unified with all (dynamic) method calls that have the (static) method call statement as a "shadow" [*Masuhara et al. (2003)*] in the program code (see next section for further explanations).

Figure 5.4 shows examples for all kinds of JPDD combination relationships (i.e. for union, confinement, and exclusion). All three examples combine the same two kinds of JPDDs (i.e. a JPDD named "SelectingAMethodCall" and a JPDD named "FurtherConstraints"). Combination takes place by (implicitly) mapping elements with identical identifiers[37].

The first JPDD, named "SelectingAMethodCall", selects all method invocations of methods named "m1" which take (exactly) one argument (of any type) and which are sent by an instance of class "Sender". The second JPDD, named "FurtherConstraints", selects all method invocations of methods named "m1" which take (exactly) one argument (that argument must be an instance of class "Integer") and which are called on an instance of class "Receiver".

When JPDD "SelectingAMethodCall" is combined with JPDD "FurtherConstraints" by means of a **union relationship** (see upper left column of Figure 5.4), the selection result contains all method calls which comply to *either* of the JPDDs[38]. In Figure 5.4, these are all method calls which are shown in the center of Figure 5.4. When the JPDDs "SelectingAMethodCall" and "FurtherConstraints" are combined by means of a **confinement relationship** (see lower left column of Figure 5.4), the selection result contains all method invocations which comply to *both* of the JPDDs. In Figure 5.4, this is

[37] Note that explicit mappings may be specified, too, by annotating the combination relationship with an explicit mapping specification (see [*Stein et al. (2005)*] for further explanations).

[38] Note that for union relationships, all identified elements of the including/source JPDD which are contained in the output parameter box of that JPDD must be mapped to an exposed element of the included/target JPDD. This is to guarantee that no element remains unspecified in the final selection result (cf. [*Stein et al. (2005)*]).

only the lower method invocation shown in the center of Figure 5.4. At last, when the JPDDs are combined by means of an **exclusion relationship** (see right column of Figure 5.4), the selection result contains all method calls which comply to one JPDDs (i.e. the including JPDD "SelectingAMethodCall"), yet *not* to the other (i.e. the included JPDD "FurtherConstraints"). In Figure 5.4, this is only the middle method invocation shown in the center of Figure 5.4.

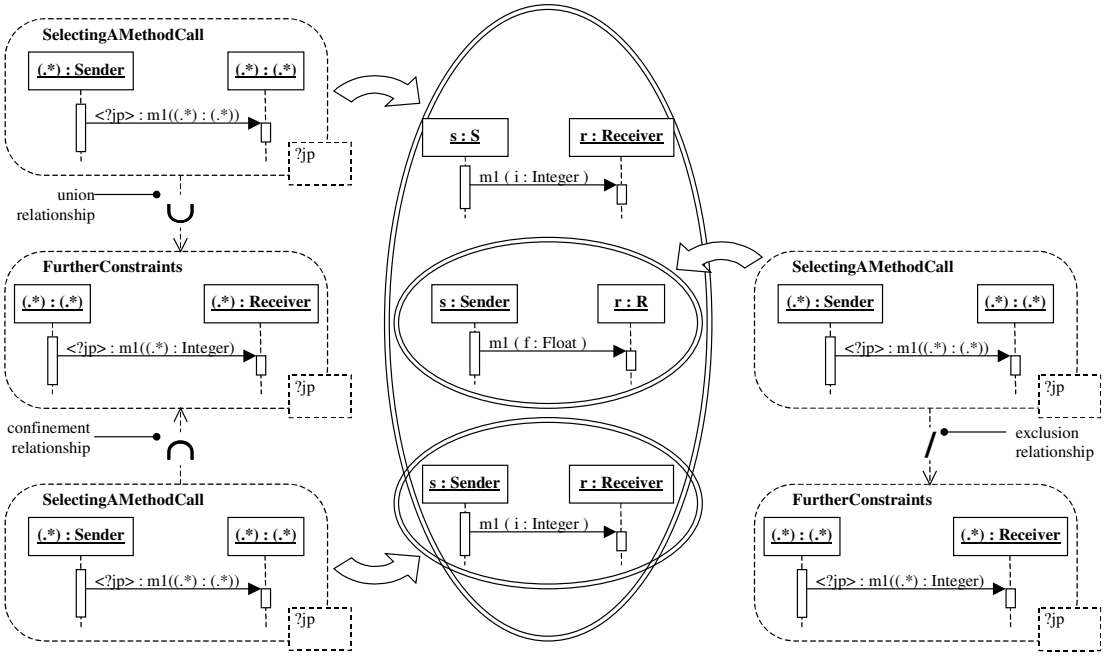


Figure 5.4 Combination semantics of JPDDs by example.

5.5 Constrainable Properties

JPDDs provide means to specify selection constraints on both the static specification of a method call (in the program code) as well as on its dynamic execution (at runtime, e.g. on the runtime context it occurs in). A means to constrain the **static specification** of a method call may confine the classes or methods where the method call must be specified in, for example. A means to constrain the **dynamic execution** of a method call may restrict the runtime types of the objects which must be involved in the method call.

With either kind of such means, JPDDs may specify selection constraints on both the structural as well as the behavioral context of a method call. An example of a selection constraint on the **structural context** would be the requirement that a particular class or object must possess a certain attribute. An example of a selection constraint on the **behavioral context** would be the requirement that a particular method call must come to pass after a certain other method call.

Table 5.1 gives an overview of the characteristics of a method call which may be constrained with help of these selection means. Figure 5.5 illustrates how the different kinds of selection constraints can be visualized with help of JPDDs.

For example, the JPDD "constrainingMethodCalls" (in the middle of the right column of Figure 5.5) represents a selection constraint on the *dynamic* and *behavioral* context of a

Table 5.1 Examples of join point properties which may be constrained with help of JPDDs.

	Static	Dynamic
Structural	Class/Method Declaration	Participating Objects/Instances
Behavioral	Definition of Method Call Statements in Method Bodies	Preceding/Subsequent Method Calls/Invocations

join point. It selects combinations of one method call and two objects. That is, it selects method calls to methods whose name is "print" together with their sender object and their (one and only) argument object. The argument object may be of arbitrary type (as indicated by the regular expression ".*"). The sender object, however, must be an instance of class "D", and the receiver object of the method call must be an instance of class "B". The selection constraint can be evaluated by observing the dynamic execution of a program (e.g. by monitoring all invocations of method "print" and collecting the participating instances, i.e. "&200F13" in Figure 5.5; see middle of left column).

Note that the JPDD "constrainingMethodCalls" includes the selection constraints specified by the other two JPDDs shown in Figure 5.5 (named "constrainingClasses" and "constrainingObjects") using confinement relationships. This means that all combinations of method calls and objects which are selected by JPDD "constrainingMethodCalls" must also comply to the selection constraints specified by JPDD "constrainingClasses" and by JPDD "constrainingObjects" (cf. previous section 5.4).

The JPDD "constrainingObjects" (at the bottom of the right column of Figure 5.5) represents a selection constraint on the *dynamic* and *structural* context of a join point. It augments the selection constraints specified by JPDD "constrainingMethodCalls" with the following additional constraint: all sender objects of the selected method calls must (be instances of a class named "D" and they must) possess an attribute named "att2" (of any type) which holds the value "hello". Such selection constraints can be evaluated by investigating the heap of a program (e.g. in Figure 5.5 by looking at the two instances "&200F13" and "&FF2345" of class "D" and their attributes "att2" which refer to instances "&1234AB" and "&337744" of class "String" which both store the required value "hello"; see bottom of left column of Figure 5.5).

Similar to JPDD "constrainingMethodCalls", JPDD "constrainingMethodCallStatements" (at the top of the right column of Figure 5.5) defines selection constraints on the *behavioral* characteristics of a join point. However, in contrast to JPDD "constrainingMethodCalls", it refers to the *static* specification of the join point (or rather its "shadow" in the program code; cf. [*Masuhara et al. (2003)*]). The JPDD augments the selection constraints specified by JPDD "constrainingMethodCalls" as follows: all selection method invocations must result from the execution of a method call statement which is defined in a class named "C" and which invokes a method named "print" (taking an arbitrary number of parameters) on an object whose declared type is "B". The selection constraint can be evaluated by looking at the implementation of class "C" in the program code (e.g. at the method call statement "b.print(this)" in method "callB" – whose signature declares "b" to be of type "B" – in class "C" in Figure 5.5, top of left column).

Note that the problems outlined in Chapter 3 strongly relate to the incapabilities of the existing approaches to represent selection constraints on the *dynamic* and *behavioral* context

of a join point. Consequently, this thesis concentrates on the specification of selection constraints on the dynamic and behavioral context of join points. It must be emphasized, however, that it is usually necessary to specify selection constraints on the static and/or the structural context of a join point, as well, in order to appropriately designate the desired set of join points that shall be adapted by an aspect. That is why JPDDs provide appropriate means to express such selection constraints, which are briefly introduced in subsection 5.8.

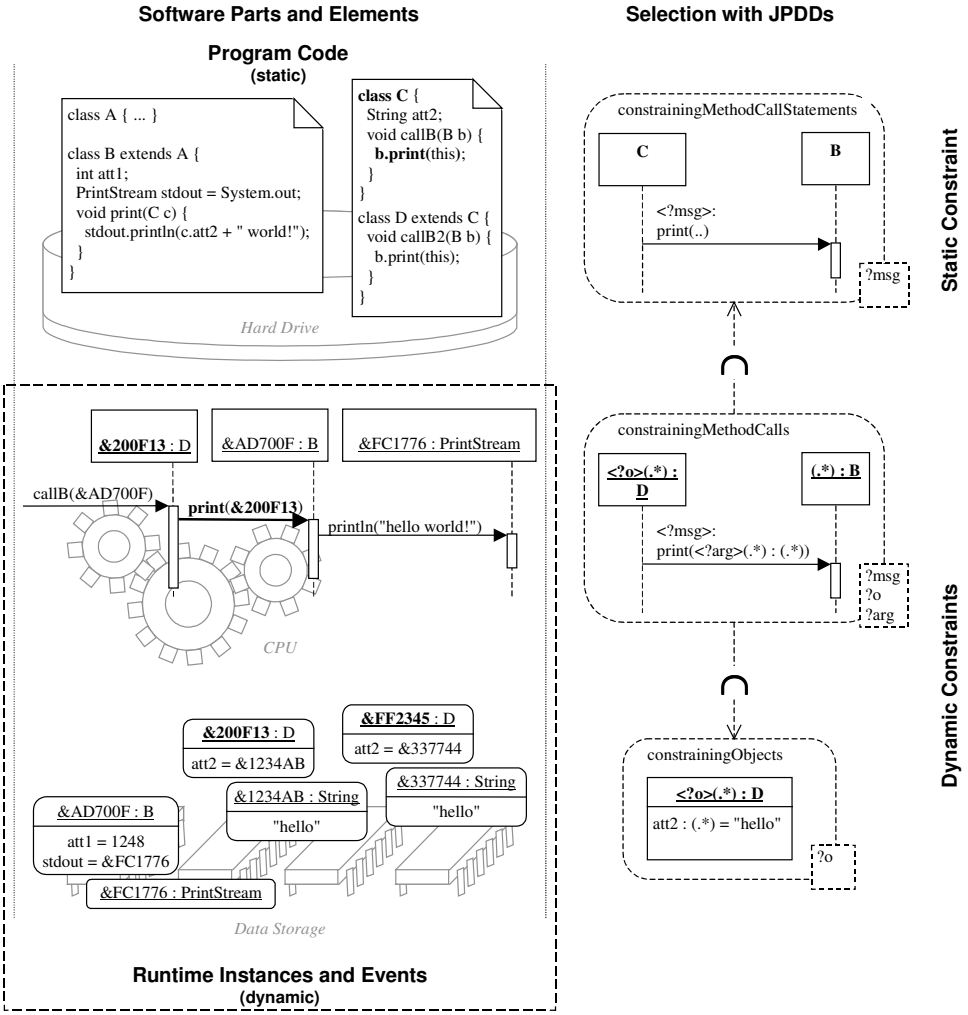


Figure 5.5 Specifying selection constraints on join point properties of different kinds.

5.6 Deviation Means

5.6.1 Regular Expressions and Wildcards

JPDDs provide various deviation means which permit to limit or constrain the values of join point properties to a range of values rather than to a distinct value. As a result, all elements selected by a JPDD must have properties whose values lay in the specified range. Examples of deviation means are regular expressions and the ".." wildcard. In a JPDD, any string representing a name of an element is a regular expressions, by default. Accordingly, all elements which are selected by a JPDD must have names which comply to those regular expressions. The regular expression ".*" may be used in JPDDs to refer to entities of

arbitrary name (i.e. the special pattern ".*" basically denotes an all-quantifier). The ".." wildcard may be used in parameter lists in order to abstract from an arbitrary number of parameters. As a consequence, parameter lists of elements which are selected by a JPDD may have more parameters (at the position of the ".." wildcard) than specified in the JPDD.

The wildcards supported by JPDDs are available in many aspect-oriented programming languages, too, e.g. in AspectJ, AspectC++, JAsCo [*Vanderperren et al. (2005)*], Aquarium [*Wampler (2008)*], Perl Aspect [*Kennedy et al. (2009)*], etc. Some of these programming languages only provide all-quantifiers (denoted by an asterisk "*" or a percent sign "%", for example) to abstract from an arbitrary number of characters in element names. Other languages however, such as Perl Aspect or Aquarium, offer the full power of regular expressions to specify such characteristics – such as JPDDs do, too.

For example, the JPDD shown in Figure 5.6 contains two lifeline symbols whose names are specified as regular expressions (".*Sender" and ".*Receiver"). Accordingly, the JPDD only selects method invocations which are sent and received by objects which are instances of classes whose names end with "Sender" and "Receiver", respectively. The all-quantifying regular expression ".*" is used to denote that the aliases of (all of) the objects are irrelevant for the selection. The ".." wildcard is used in the method signature of the method symbol in order to denote that selected method calls may be provided with an arbitrary number of additional arguments (i.e. none, one, or more) between the (required) arguments, which must be instances of the classes "Integer", "Float", and "String", respectively. Figure 5.6 shows both a matching and a non-matching class on the right side.

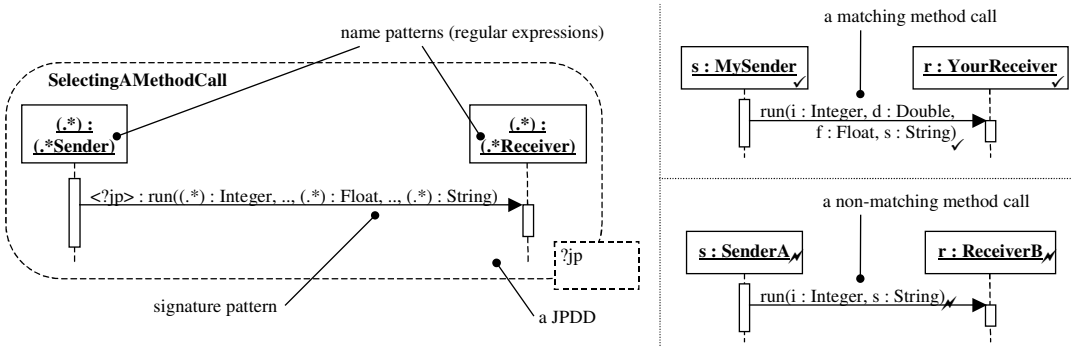


Figure 5.6 Regular expressions and ".." wildcard.

5.6.2 Path Expressions

Another way to relax selection criteria is the possibility to specify paths, e.g. along the inheritance tree, the object graph, or – in particular – along the call graph. In JPDDs, paths are visualized as indirect relationships (⇗). In contrast to direct relationships, indirect relationships permit that elements participating in the relationship are not immediate neighbors. A multiplicity may be given to restrict the number of relationships (e.g. the number of inheritance relationships, object associations, or method calls) that must be traversed to reach one participating element from the other. In control flow-oriented join point selections, an indirection symbol may also be used to denote indirect activation bars (⇵). The meaning of indirect activation bars is similar to indirect relationships in the sense that two method calls initiated from the same activation bar, which are separated by an indirect symbol, do not need to be immediate successors.

Path expressions are provided by JPDDs in order to accommodate the feasibility to specify join point selections based on paths, which is possible in many aspect-oriented programming languages. For example, AspectJ – as well as AspectC++, JAsCo, Alpha and many more languages – permit to specify join point selections which constrain (particular properties of) paths along the dynamic *call graph*; Declarative Event Patterns (DEP) [*Walker & Viggers (2004)*], JAsCo, and Tracematches permit to specify selection constraints on paths (or "traces") in the *execution history* of a program; Path Expression Pointcuts (PEP) [*Al-Mansari & Hanenberg (2006)*] and Alpha permit to specify selection constraints on paths over *object graphs*; AspectJ – and most of the previously mentioned languages – permit to specify selection constraints on paths along the *type hierarchy*; and Traversal Strategies [*Lieberherr (1995)*] as well as CARMA (formerly known as Andrew) [*Gybels & Brichau (2003)*] permit to specify selection constraints on paths along the *class hierarchy*. Indirect activation bars relate to "safe parts" (as opposed to "strict parts") in the semantic-based weaving approach presented in [*Klein et al. (2007)*]. Note that JPDDs support all of the previously mentioned kinds of path. For illustration purposes, this section focuses on paths along the dynamic call graph, only. Most of the other kinds of paths will be exemplified in the oncoming subsections, though.

Figure 5.7 gives an example (which is adopted from an example presented in [*Soares et al. (2002)*]). The JPDD makes use of an indirect message as well as indirect activation bars. The indirect message is used to denote that method "search" may be invoked by some other method which was (maybe transitively) invoked by method "doPost". This means that the invocation of method "search" must occur on a higher call stack level than the invocation of method "doPost" (it also means that method "doPost" must still be "active", i.e. must still be on (a lower level of) the call stack, when the invocation of method "search" occurs). The lower bound "0" of multiplicity "[0..*]" indicates that the method "search" may also be invoked by method "doPost" immediately (i.e. it is not strictly necessary that another method call must have occurred in between, which then calls method "search" transitively).

An indirect activation bar is used in Figure 5.7 to denote that the method call to method "search" may not be the first one issued by the current method (i.e. on the current call stack level). Likewise, another indirect activation bar is used to denote that the method call which leads to the invocation of method "search" may not be the first one issued by method "doPost".

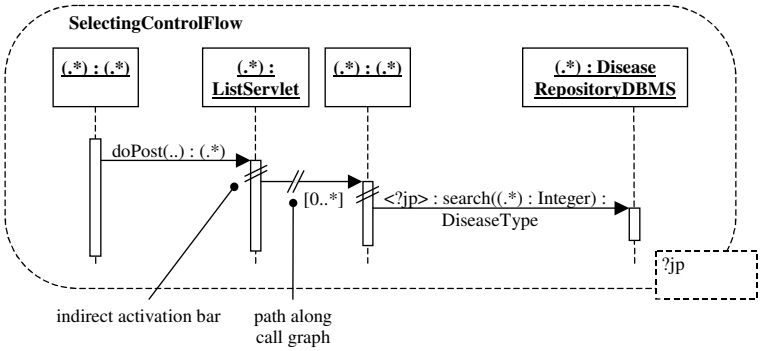


Figure 5.7 Paths and indirect activation bars.

In summary, the JPDD shown in Figure 5.7 selects all method calls of method "search" which take one instance of class "Integer" as the (one and only) argument and which return

an instance of class "DiseaseType". These method calls must come to pass in the control flow of a method call to method "doPost" which takes an arbitrary number of arguments (denoted by the ".." wildcard) and which returns an instance of arbitrary type. This means that the method call to method "doPost" must not have terminated yet (i.e. it must still be "active"), and that – correspondingly – the method call to method "search" must occur at a higher call stack level than the method call to method "doPost". Nevertheless, as indicated by the indirect activation bars, there may be other method calls which may have been invoked (and terminated) on intermediate call stack levels before method "search" is reached. As an additional constraint, the JPDD specifies that the receiver object of the call to method "search" must be an instance of class "DiseaseRepositoryDBMS", and that the receiver object of the call to method "doPost" must be an instance of class "ListServlet".

5.7 Conceptual Views

JPDDs offer various notational means to specify join point selection constraints on program behavior, each suited to express a particular conceptual view on program execution (cf. [*Stein et al. (2006)*]). For example, one subset of JPDD symbols refers to a control flow-oriented view on program execution. That is, it emphasizes how program control is passed on (and subsequently returned) from one method to another. Another subset of JPDD symbols refers to a workflow and data flow-oriented view on program execution. That is, it emphasizes the execution steps which are performed to fulfill a particular task, as well as the data which is involved in these steps of this task. Finally, a third subset of JPDD symbols refers to a state and transition-oriented view on program execution. That is, it emphasizes that objects show different behavior depending on the state their are in.

The provision of multiple notations to specify join point selections has been inspired by the observation that join point selections may realize different conceptual models of program execution. The original AspectJ perspective on program execution, for example, is that of synchronously interacting objects, i.e. – to be more precise – that of synchronous method invocations making up a call graph (cf. [*Kiczales et al. (2001)*, *Hilsdale & Hugunin (2004)*]); [*Douence et al. (2001)*] have coined the notion of Monitor-Based AOP (later called Event-Based AOP; EAOP) and introduced Stateful Aspects [*Douence et al. (2004)*]; finally, [*Masuhara & Kawauchi (2003)*] have highlighted the need of a data flow-oriented view on program execution. At the same time, it has been observed that all of these conceptual view on program execution have been around in conventional software development for a long time, together with suitable notations to express such conceptual views. None of these notations has been capable of expressing the specification of aspect-oriented join point selections, though. JPDDs adopt and extend these notations, thus providing appropriate means to express aspect-oriented join point selections in a way which accommodates and maintains the conceptual view of program execution underlying the given join point selection.

The notational means to express these different conceptual views are based on different (sets of) UML symbols. All of them adopt the JPDD-specific means, though, which have been introduced in the previous sections (i.e. identifiers, regular expressions, wildcards, and paths). The following subsections introduce the different national means and exemplify which join point selection constraints are best represented with which notational means. The introduction/discussion begins with the notational means that have been used in the

previous sections to introduce the JPDD-specific means, i.e. with the control flow-oriented selection means.

5.7.1 Control Flow View

The notational means referring to a control flow-oriented view on program execution are based on UML interaction diagram symbols. These means are particularly suited to express selection constraints that refer to the flow of control between methods. For example, developers may require that one method must invoke another method, or that one method must come to pass in the control flow of another method. Figure 5.8 shows an example. The example is adopted from [*Soares et al. (2002)*] and is about an aspect which intercepts internet accesses to an online disease repository in order to reduce the load time of complex data objects (i.e. "DiseaseTypes") when only partial information is needed. Accordingly, the join point selection (shown in Figure 5.8, which is identical to the one shown in Figure 5.7) selects all accesses to the disease repository (issued via method "search") which come to pass in the control flow of a servlet request (issued via method "doPost").

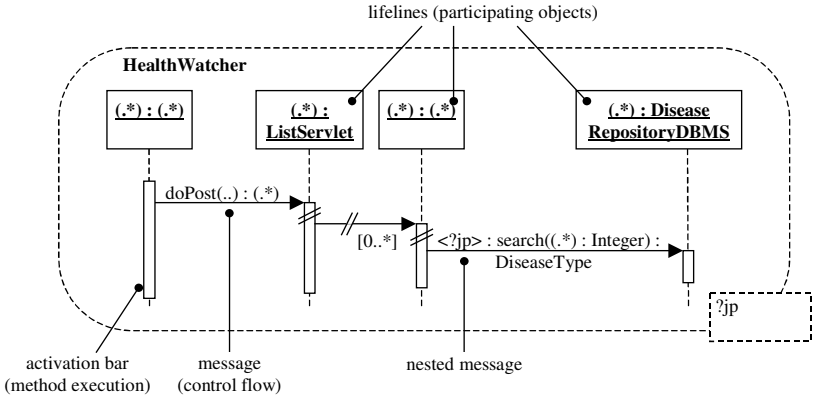


Figure 5.8 Expressing a control flow-oriented join point selection.

Note how the control flow-oriented representation of the join point selection shown in Figure 5.8 emphasizes the chronological relationship of the two method calls of method "search" and method "doPost", which denotes a key characteristic of the join point selection. The representation clearly indicates that it is essential for the join point selection that the one method call occurs while the other method call is still "active", i.e. "executing". That is why interaction diagram-based JPDDs are considered to be particularly suited to express control flow-oriented join point selection constraints: they outline how control is passed from one instance to another by means of method invocations; they render the chronological dependencies between method calls, and indicate how each method is invoked in the dynamic context of another; finally, the activation bars on the lifelines of each object indicate that each method remains active until the termination of the subsequent methods.

5.7.2 Data Flow View

The situation is different if the control flow-oriented selection means are used to express selection constraints that focus on data flow. For example, Figure 5.9 illustrates a join point selection which is taken from [*Alhadidi et al. (2009)*] and which is concerned

with the interception of database queries that include strings from untrusted origins (e.g. from the internet). The objective of the corresponding aspect is to prevent the execution of malicious queries on the database.

Figure 5.9 illustrates what the join point selection looks like using control flow-oriented join point selection means: the join point selection selects all invocations of method "executeQuery" (issued on an instance of class "Statement") which take the return value of an (earlier) method invocation of method "getParameter" (issued on an instance of class "HttpServletRequest") as argument. Note how the JPDD makes use of multiple indirect messages and indirect activation bars to denote that the method invocation of method "getParameter" must occur "some time" before the method invocation of method "executeQuery". Furthermore, note how the JPDD makes use of identifiers (i.e. "<?val>") to denote that the argument of method "executeQuery" must be the same object as the return value of method "getParameter".

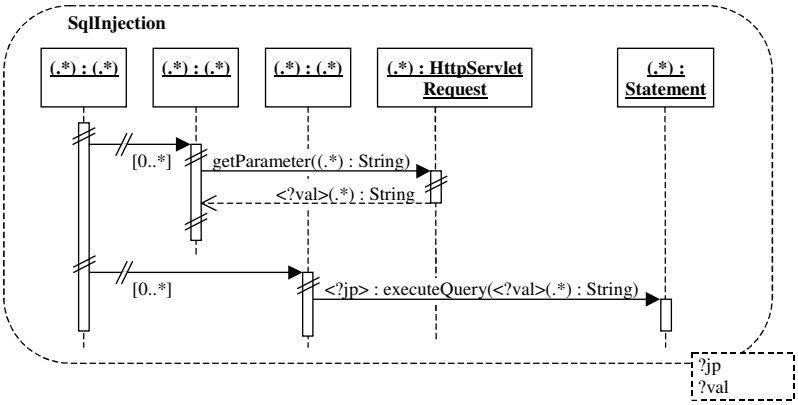


Figure 5.9 Inferior representation of a data flow-oriented join point selection.

Taking a critical look at the JPDD shown in Figure 5.9, it must be recognized that – although the chronological dependency between the invocation of method "getParameter" and of method "executeQuery" is properly visualized – the key selection constraint, i.e. the data flow of object "<?val>", is not sufficiently highlighted: one must carefully study the diagram in order to discover that the object "<?val>" being returned by method "getParameter" and the object "<?val>" being passed to the "executeQuery" method must be the same object.

From a modeling perspective, this is not satisfactory. What we would like to have is an explicit visualization of object "<?val>", as well as of the different ways it is involved in each method. It should be easy to recognize that object "<?val>" is (both) output from method "getParameter" and input to method "executeQuery". In interaction diagrams, however, input and output parameters are rendered as "annotations" to messages only. Hence, they are incapable of stressing the fundamental significance of input and output parameters to the selection result of data flow-based queries. The use of interaction diagram-based JPDDs to represent selection constraints pertaining to data flow must therefore be considered inappropriate.

To resolve this shortcoming, JPDDs offer a supplementary set of symbols which permits to emphasize selection constraints that require the involvement of data in multiple system events. This set of symbols is based on UML activity diagram symbols. Figure 5.10 gives an example:

Figure 5.10 illustrates how the join point selection constraint shown in Figure 5.9 may be expressed using an activity diagram-based JPDD. Figure 5.10 shows two call action symbols: one call action symbol represents the invocation of method "getParameter", and the other call action symbol represents the invocation of method "executeQuery". The actions are connected to each other by an indirect control flow symbol (which is annotated with a multiplicity of "[1..*]"), which means that multiple actions (i.e., at least one, in this case) may take place between these two actions. Analogously, indirect control flow symbols are used to connect the call actions with the initial state symbol and the final state symbol in the JPDD, which means that method "getParameter" does not need to be the first action and method "executeQuery" does not need to be the last action in the workflow. The actual data flow is represented using object flow symbols: action "getParameter" returns an instance "<?val"> of class "String" as output parameter, which is then passed to action "executeQuery" as an input parameter. Other object flow symbols are used to depict the involvement of the target objects and of the argument object in the call actions.

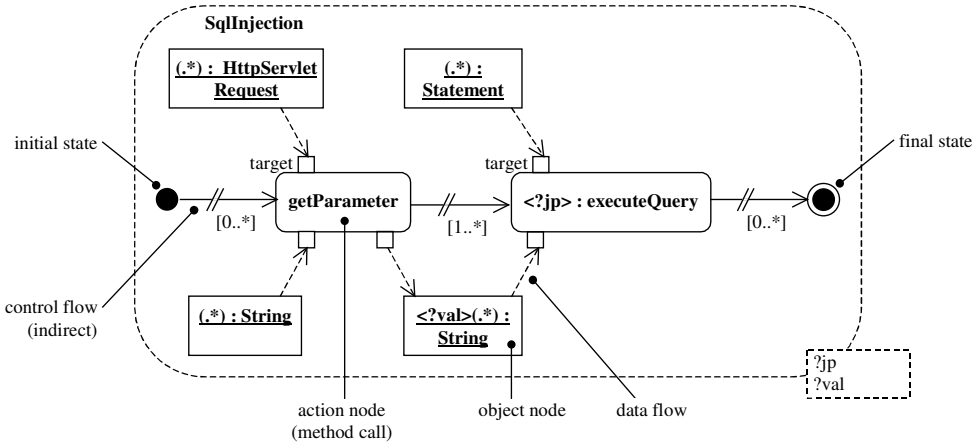


Figure 5.10 Expressing a data flow-oriented join point selection.

When comparing the control flow-oriented representation of the join point selection shown in Figure 5.9 with the data flow-oriented representation shown in Figure 5.10, it can be observed that the data flow-oriented representation puts an emphasis on the data dependency between method "getParameter" and method "executeQuery", which is a key characteristic of the join point selection. As a result, it is easier for developers to observe that the return value of method "getParameter" must be an argument of method "executeQuery". That is why activity diagram-based JPDDs are considered to be particularly suited to express data flow-oriented join point selection constraints: they put a main focus on the order of actions as well as on the data involved, and thus emphasize the dependencies that must exist between different actions as well as between actions and data. Note, though, that other than interaction diagram-based JPDDs, they do not consider how program control is handled over from one action to another (e.g. from method "getParameter" to method "executeQuery").

5.7.3 State View

Similar problems as the ones described in the previous subsection arise when control flow-oriented join point selection means are used to express state-based join point selections.

For example, consider a persistency aspect which traps accesses to transient representations of deleted persistent objects that have not yet been collected by Java's garbage collector (the example is adopted from [*Rashid & Chitchyan (2003)*]). Figure 5.11 illustrates what a corresponding join point selection looks like using the control flow-oriented selection means: the JPDD selects all invocations of methods whose names begin with "set", "get", or are "toString", and which are issued on instances of classes whose names begin with "Persisted". In addition to that, the selected method invocations need to occur after an invocation of method "delete", which is issued on the very same instance as the aforementioned method invocations (this is denoted by having the symbols of both method invocations point to the same lifeline).

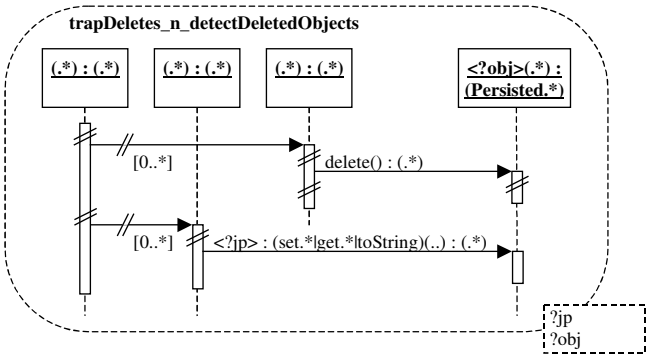


Figure 5.11 Inferior representation of a state-based join point selection.

Investigating the appropriateness of the JPDD shown in Figure 5.11, it can be observed that the control flow-oriented representation of the join point selection is well suited to emphasize that the "delete" method must be invoked "some time" before the access methods (denoting the join point). However, it fails to emphasize the effects that such method invocations have on the system or object state. In this case, for example, the diagram does not emphasize that the object receiving the "delete" method call is deemed to transition to some state "deleted", in which it is not supposed to reply to any more requests and to fulfill any more method invocations.

From a modeling perspective, this must be considered inappropriate since the selection criteria in the original problem was to select objects that have reached state "deleted" – rather than to select objects that have received a message invoking their "delete" method. We would like to have an explicit visualization of that state change: a visualization that emphasizes the pivotal importance of this criterion to the join point selection (and to the aspectual adaptation that follows). Since interaction diagram-based JPDDs do not provide such means to effectively indicate a state change of an object, they are not suited to point out the particular relevance of these object states to a given join point selection.

To rectify this deficiency, JPDDs provide a special set of symbols which are specifically suited to express selection constraints that refer to object or system states as well as to object or system transitions. This set of symbols is based on UML state chart symbols.

Figure 5.12 illustrates what a state chart-based JPDDs looks like for the sample join point selection described above. The JPDD depicts an instance of a class whose name begins with "Persisted". That instance has a state chart which outlines the relevant behavior of the instance which eventually leads to the selection of a join point. Accordingly, the state chart defines two states which are connected by a transition. That transition is triggered by an invocation of method "delete". The source state of that transition is connected to the

initial state, and thus represents the state which the object is considered to be in (seen from the perspective of the aspect) as soon as it is created. The target state of the transition is given an identifier, i.e. "<?deleted>" (the intention of doing so is to facilitate the reading as well as the explanation of the join point selection). Once the instance is in that state "<?deleted>", the JPDD intercepts all further method invocations – i.e. those invoking methods whose names begin with "set", "get", or are "toString" – as join points. Note that the selected method invocations have no effect on the object's state; consequently, they are depicted as a (trigger to a) self-transition. Note further that the states defined in the JPDD are considered aspect-specific, i.e. they do not refer to unique object/program states in the base program. Instead, they are considered to emerge from the mere occurrence of the events which trigger the transitions they are connected to[39]. Accordingly, state "<?deleted>" is characterized solely by the events triggering the transitions it is connected to; no restrictions are made concerning its name, or its entry, exit, and do actions, etc.

Investigating the suitability of the state chart-based JPDD shown in Figure 5.12, it may be observed that the representation emphasizes the relevance of the state change to the join point selection. Other than the control flow-oriented representation shown in Figure 5.11, it abstracts from the general system behavior as a sequence of actions and focuses on the effects of such actions with respect to the system state. Accordingly, the JPDD in Figure 5.12 identifies and highlights the consequence of the "delete" method call, which is a state transition from any state (".*") to the state "<?deleted>". By highlighting this key characteristic of the join point selection, it should be easier for developers to estimate the – state-dependent – objectives and the results of this join point selection.

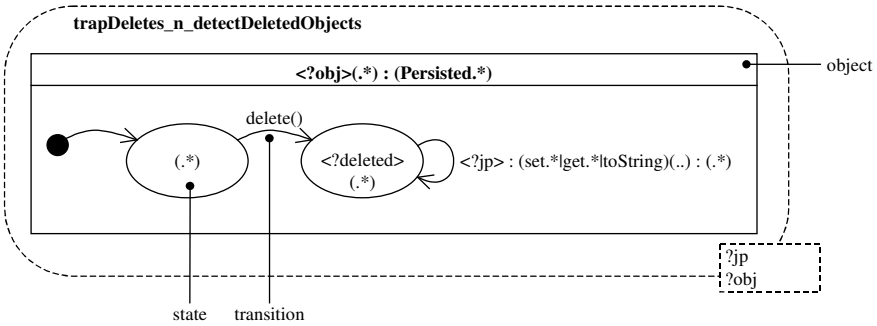


Figure 5.12 Expressing a state-based join point selection.

That is why state chart-based JPDDs are considered to be particularly suited to visualize join point selections relying on system state, object state, and their transitions: developers are able to highlight the significant effect that state changes have on a join point selection.

5.7.4 Concluding Remarks

Note that the different subsets of JPDD symbols are not equally expressive (see Table 5.2). For example, the data flow-oriented symbols and the state transition-oriented symbols are not capable of expressing selection constraints on nested actions[40]. Instead, they imply

[39] The all-quantifier ".*" in the name pattern of the states is intended to remind the reader of this fact (i.e. that states map to "anything"); name pattern of states generally do not affect the join point selection result.

[40] Note that UML activity diagrams offer the possibility to indicate a call of an (sub)activity by placing a rake symbol in the bottom corner of a call action; the called (sub)activity is then represented by an activity diagram of its own; as an alternative, some UML modeling tools offer the possibility to place the called (sub)activity diagram right inside the

that a given action has always terminated (successfully) before the next action occurs[41]. In contrast to that, control flow-oriented symbols may explicitly specify that one method invocation must still be on the call stack while another method invocation occurs (control flow-oriented symbols always refer to particular (relative) call stack levels). On the other hand, state transition-oriented symbols may be used to specify that the occurrence of a particular method (maybe temporarily) prohibits the selection of another method – which is not possible with control flow-oriented and data flow-oriented symbols[42]. For example, the lower left JPDD in Figure 5.13 specifies that an invocation of method "quit" makes the system leave state "testing" (see gray colored transition in Figure 5.13), and thus suspends the selection of join points until a new invocation of method "setup" occurs. Finally, state transition-oriented symbols may not be used to specify that two or more execution steps must occur immediately after each other (which is possible with both control flow-oriented and data flow-oriented symbols).

Table 5.2 Expressiveness of different JPDDs.

Expressing selection constraints on...	Interaction diagram-based JPDDs	Activity diagram-based JPDDs	Start chart-based JPDDs
...action sequences	✓	✓	✓
containing...			
...nested actions	✓	✗	✗
...prohibitive actions	✗	✗	✓
...alternative action sequences	✗	✗	✓
...(immediate) successor actions	✓	✓	✗

In simple cases, it may be possible to express a join point selection with either means, though. Then, it is a matter of perspective which selection means should be used. The perspective depends on the key characteristic that shall be highlighted (be it control flow, data flow, or states and transitions). An example is given in Figure 5.13. Both of the lower JPDDs express the same join point selection (provided that the gray colored transition in the lower left JPDD is discarded): the state chart-based JPDD on the left emphasizes that the join point selection only occurs if the system is in state "<?testing>", while the activity diagram-based JPDD on the right emphasizes that the selection only occurs if the argument of the method call denoting the join point (i.e. a method call to method "run") has also been an argument to a previous method call (i.e. to method "setup").

In summary, Figure 5.13 illustrates the three different conceptual views on program execution which are supported by JPDDs (the example is inspired by [*Lesiecki (2005b)*] and is about mock testing with help of aspect-oriented programming), together with method invocation scenarios which satisfy the JPDDs shown next to them (in each

symbol of the call action; for the time being, JPDDs chose not to adopt these capabilities because it leads to multiple (possibly nested) activity diagrams which fail to emphasize the interactions between the participating objects such as interaction diagrams do.

[41] Interception of a join point occurs *before* the action is executed, though, in order to provide for the execution of `before`, `around` and `after` advice.

[42] at least not using a single JPDD; in some cases, this may be expressed with help of combination relationships.

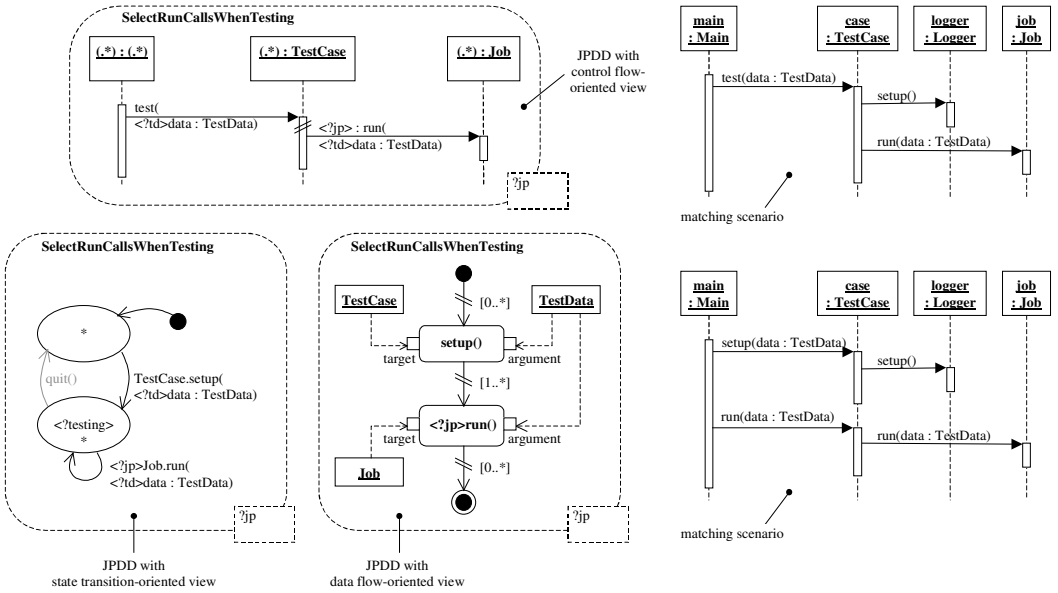


Figure 5.13 Expressing different conceptual views on a join point selection.

scenario the invocation of method "run" would be selected). The JPDD at the top of Figure 5.13 reflects a control flow-oriented view on program execution. It makes use of UML interaction diagram-based symbols to emphasize that the selected method call of method "run" must come to pass in the control flow of a method call to method "test" (i.e. method "test" must still be "active", i.e. on the call stack, for the selection to take effect). The lower right JPDD in Figure 5.13 reflects a data flow-oriented view on program execution. It makes use of UML activity diagram-based symbols to emphasize that the same "TestData" object must be passed as an argument to both method calls (i.e. to method "test" and to method "run"). Finally, the lower left JPDD in Figure 5.13 reflects on a state transition-oriented view on program execution. It makes use of UML state chart-based symbols to exemplify that the system must be in state "testing" for the join point selection to take place (and that the system is considered to be in state "testing" once the "setup" method has been called).

5.8 Static and Structural Constraints

Apart from selection constraints on the runtime behavior of a program, aspect-oriented join point selections frequently include selection constraints on the program code that specifies this runtime behavior. To express such selection constraints, JPDDs provide an extra set of symbols which are based on UML class diagrams (the symbols may be combined with the JPDD-specific deviation means that have been introduced in section 5.6). It is subject of this subsection to introduce these symbols and to illustrate their usage. Note that this introduction is kept rather short since the mere goal of this subsection is to demonstrate that JPDDs provide all necessary means to express the most common selection constraints of prevailing aspect-oriented join point selections. The major focus of this thesis is on (the comprehension of) the selection constraints on dynamic and behavioral properties of a program. That is why the notational means to specify selection constraints on the static properties of a program are not taken into account in the remainder of this thesis.

Figure 5.14 illustrates how JPDDs may express selection constraints on the static and dynamic properties of a method call. The notational difference between the two is subtle, yet significant: the right JPDD makes use of *object symbols* to represent the interacting instances, which denotes that the JPDD refers to dynamic method call events (which occur at runtime); the left JPDD makes use of *class symbols* to represent the corresponding entities, which denotes that the JPDD refers to static method call statements (which are defined in the program code). Accordingly, the right JPDD selects all method call events that are initiated by an instance of class "MySubClass" at runtime (regardless of the method being called and the instance being addressed), while the left JPDD selects all method call statements which are specified in class "SuperClass" in the program code. Provided that "MySubClass" is a subclass of "SuperClass", the combination of these JPDDs (as illustrated in Figure 5.14) selects all method invocations issued by an instance of class "MySubClass" which are defined by its super-class "SuperClass" (such method invocations occur, for example, when an instance of class "MySubClass" calls a method of its super-class "SuperClass" using the "super" keyword).

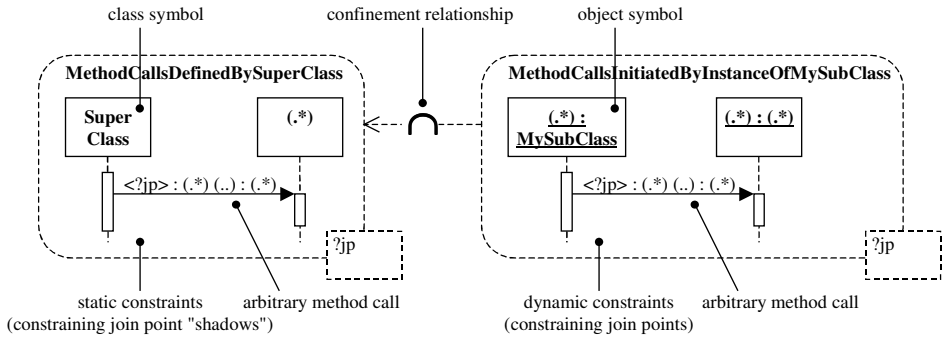


Figure 5.14 Combining selection constraints on static and dynamic properties of a method call.

Apart from selection constraints on the static specification of the program behavior, JPDDs may also express selection constraints on the static specification of the data structures which participate in that behavior. Figure 5.15 illustrates an example. The JPDD depicts a class which is recursively contained in several packages, and which is connected to another class by means of a generalization relationship. A regular expression is used to specify the name of the generalized (or inherited) class (i.e. "Super.*"). And a ".." wildcard is used (in the name compartment of the middle package) to denote a recursive package structure of arbitrary depth (consisting of packages of arbitrary names). An indirection symbol is used to denote a transitive generalization relationship of arbitrary length (which means that a path exists along the generalization hierarchy (i.e. the inheritance tree) consisting of an arbitrary number of intermediate generalization relationships, which leads from the specialized (or inheriting) class to the generalized (or inherited) class).

Ultimately, the JPDD shown in Figure 5.15 selects all classes with name "MyClass" which are contained in a package named "MyPackage", which – in turn – needs to be (recursively) contained in a package named "TopPackage". In addition to that, the class needs to be a transitive descendant (i.e., not necessarily a direct child) of a class whose name matches the regular expression "Super.*"[43]. Figure 5.15 shows both a matching and a non-matching class on the right side.

[43] Note that no constraint is made in the JPDD concerning the package containment of that ancestor class.

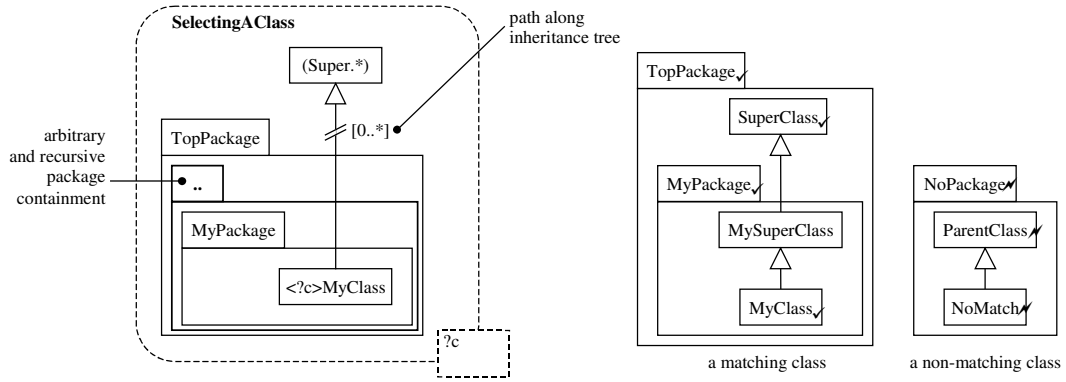


Figure 5.15 Selection constraints on the (static) specification of a class.

Figure 5.16, finally, illustrates how the selection constraints on the runtime behavior of a program may be combined with the selection constraints on the static specification of the data structures that participate in that behavior. To do so, the JPDD makes use of a "multi-JPDD separator", (which is depicted as a dotted line and) which may be used to combine selection constraints that are expressed using different sets of JPDD symbols in a single JPDD. A multi-JPDD separator is semantically equivalent to a confinement relationship, which means that all (identified) elements in one part of the JPDD must additionally satisfy the selection constraints outlined in the other part(s) of the JPDD. In case of the JPDD shown in Figure 5.16, this means that all (identified) instances which participate in the object interaction outlined in the right part of the JPDD must additionally satisfy the inheritance constraints outlined in the left part of the JPDD. Thus, the receiver object of method "post" must be an instance of a subclass of class "ListServlet", and the argument object of method "search" must be an instance of a subclass of class "SearchCode".

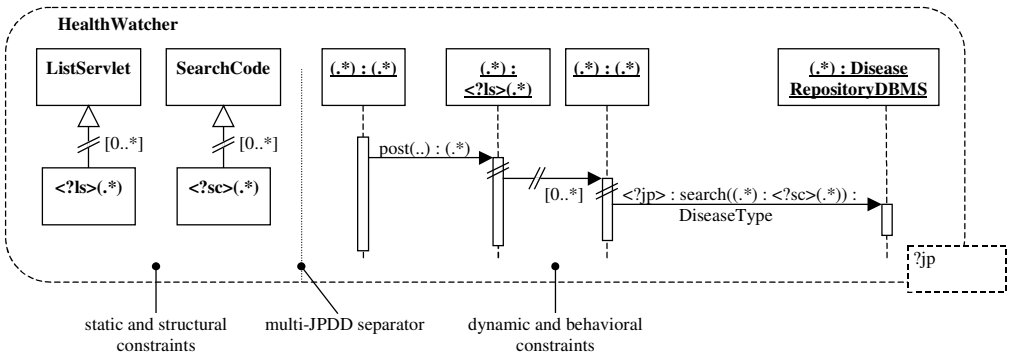


Figure 5.16 Selection constraints on structural and behavioral properties in a single JPDD.

Note that JPDDs also support the specification of selection constraints on the runtime values of the data structures that participate in the runtime behavior. To do so, JPDDs adopt the symbols of UML object diagrams. Figure 5.17 (right side) shows an example: the JPDD depicts an object symbol which contains two attribute symbols (i.e. attribute slots) that possess the values "Hello World" and "123". Accordingly, the JPDD selects all instances of class "MyClass" which contain two attributes, named "att1" and "att2", whose values are "Hello World" and "123", respectively.

As a specialty, JPDDs also permit to combine the object diagram-based notation with the class diagram-based notation in a single (class or object) symbol in order to express

selection constraints on both the runtime values of a data structure as well as on its specification in the program code. An example is given in Figure 5.17 (left side): the JPDD requires – in addition to the aforementioned constraints – that attribute "att1" must be an instance of class "String" and that attribute "att2" must be an instance of class "Integer"; furthermore, class "MyClass" (which all selected objects must be an instance of) must possess an operation "op1" which takes exactly one argument named "val1" of type "Integer". The combination of these selection constraints in a single symbol is equivalent to their specification in distinct JPDDs, which are subsequently combined using a confinement relationship (as illustrated by the center and right JPDDs in Figure 5.17).

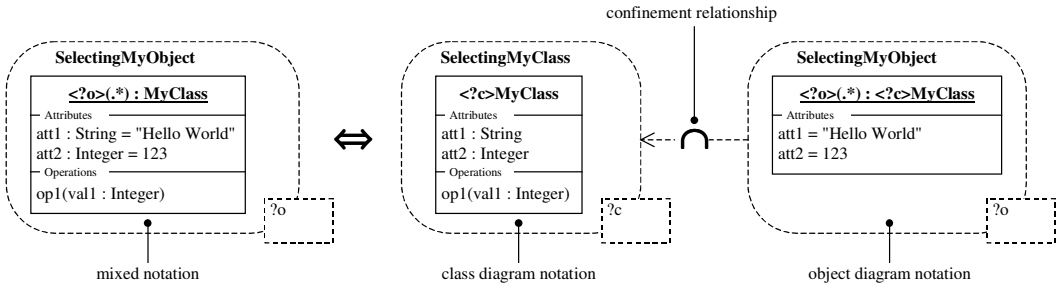


Figure 5.17 Selection constraints on static and dynamic properties of a data structure.

5.9 Examples

Having introduced the JPDD-specific means to specify join point selections in the previous sections, this section illustrates their usage with help of examples. The primary motivation of these examples is to illustrate the definition of selection constraints on the dynamic and behavioral properties of join points.

5.9.1 Interaction Diagram-Based JPDDs

The first example is adopted from [*Lesiecki (2005b)*] and is about an aspect-oriented implementation of the decorator pattern [*Gamma et al. (1995)*] which augments a file input stream with a progress monitor. To do so, the aspect requires a reference to a GUI component which the monitor dialog can be tied to. The aspect retrieves that reference from the call stack of the decorated method, which – as a selection constraint – must be invoked in the control flow of (a method being called by) such GUI component. Correspondingly, the join point selection of the aspect is represented with help of an interaction diagram-based JPDD in Figure 5.18:

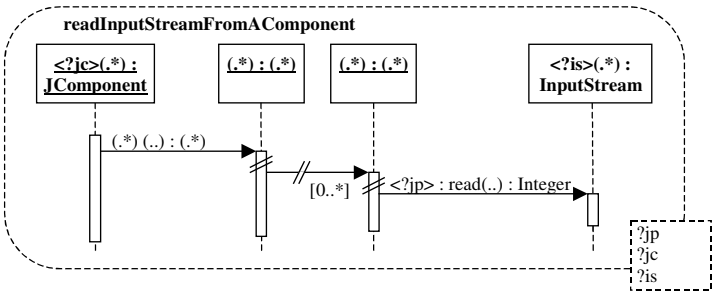


Figure 5.18 Selecting read accesses of an input stream in a graphical environment.

The join point selection selects all invocations of methods whose name is "read" and which take an arbitrary number of arguments (".."). These method calls may be initiated by an instance of any class (".*"). Yet, they must be targeted at an instance of class "InputStream". Furthermore, they must occur in the control flow of a method call to some method named "readFile" which takes an arbitrary number of arguments (".."). That method call must be issued by an instance of class "JComponent", and may be addressed to an instance of any class (".*"). The join point selection exposes the instance of the "JComponent" (which initiates the control flow) together with the instance of the "InputStream" (which is monitored) to the join point adaptation by adding their identifiers (i.e. "<?jc>" and "<?is>", respectively) to the output parameter box of the JPDD.

5.9.2 Activity Diagram-Based JPDDs

The second example is about a caching aspect which caches the return values of a complex operation (i.e. the calculation of strongly connected components in a directed graph). Whenever the operation is called again with the same input parameters, the aspect shall prevent the repetitious execution of the complex operation and shall return the cached return value instead. The involvement of the same data in multiple operations is a pivotal selection constraint in this aspect. Therefore, its join point selection is represented with help of an activity diagram-based JPDD in Figure 5.19:

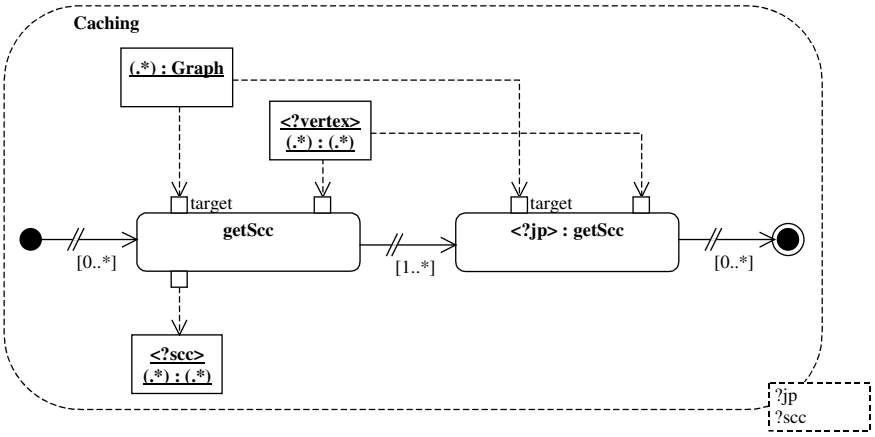


Figure 5.19 Selecting recurring invocations of complex operations.

The join point selection selects all recurring invocations of methods with name "getScc", which follow a previous invocation of a method with that same name. Both method calls must be targeted at the same instance of class "Graph" and must take the same instance (of any class (".*")) as an argument (an identifier "<?vertex>" is used to suggest that the argument refers to the vertex which the strongly connected component should be computed for). The join point selection exposes the return value of the first invocation of the method "getScc" (which may be an instance of any class (".*")) by adding its identifier (i.e. "<?scc>") to the output parameter box of the JPDD.

5.9.3 State Chart-Based JPDDs

The last example is about a state based-join point selection. That is, the join point selection selects join points only if they occur in particular object or system states. These states are determined by the method invocations which the object or the system has (or has

not) received previously. The example is adopted from [*Bockisch et al. (2005)*] and is about a simple text editor which shall be augmented with an aspect that prevents users from loosing their unsaved modifications by unintentionally closing the editor or creating a new document.

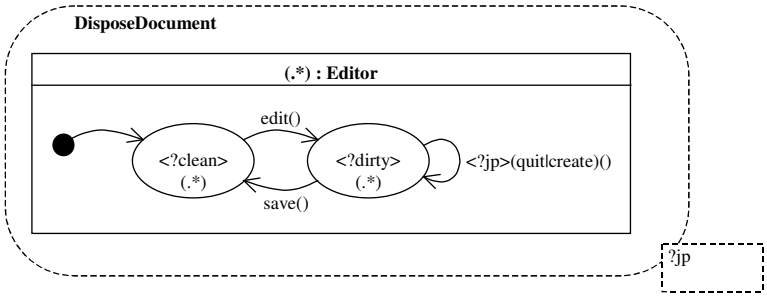


Figure 5.20 Selecting unintentional disposals of documents.

To do so, the join point selection of the aspect (shown in Figure 5.20) selects all invocations of methods whose name is "quit" or "create" which come to pass after an invocation of a method with name "edit", yet before any (subsequent) invocation of a method with name "save" (all of the mentioned method invocations must be addressed to the same instance of class "Editor"). Note how identifiers (i.e. "<?clean>" and "<?dirty>") are used to give suitable names to the states in the JPDD. This is done for indicative reasons, only. It is supposed to help readers understand the characteristics of the states in which the join point selection takes effect, and in which states it does not.

5.10 Summary

This chapter has introduced Join Point Designation Diagrams (or JPDDs, in short). JPDDs are a visual way to represent aspect-oriented join point selections. They are based on the syntax and the conceptual framework of the UML. JPDDs may constrain structural and behavioral as well as static and dynamic properties of join points (the focus of this thesis is on selection constraints on the dynamic and behavioral properties, though). To do so, JPDDs make use of wildcards, regular expressions, and path expressions. Furthermore, they offer means to express different conceptual views on program execution (such as the UML does), i.e. an control flow-oriented view, a workflow and data flow-oriented view, as well as a state transition-oriented view (apart from the static and structural view, which is not at the focus of this thesis, though). These different notational means may be used to emphasize different key selection constraints of a join point selection (such as control flow-based, data flow-based, or state-based selection constraints).

5.10.1 Outlook to Next Chapters

The next chapter discusses the suitability of JPDDs to facilitate the comprehension of complex join point selections with the help of examples. Afterwards, that suitability is evaluated with the help of an empirical experiment.

Chapter 6
Discussion

This chapter discusses how JPDDs may help to improve the comprehension problems that readers may have with textual implementations of complex join point selections (such as they have been discussed in Chapter 3). To do so, the chapter revisits each of the motivating examples presented in Chapter 3 – as well as additional examples, which are discussed in the Appendix A of this thesis – and presents a corresponding JPDD for each example. Subsequently, it compares the JPDDs to the textual implementations of those examples and elucidates why JPDDs are considered to improve the comprehensibility of those sample join point selections. The chapter concludes with a summary.

6.1 Revisiting the Motivating Examples

In this section, a JPDD is presented for each of the motivating examples presented in Chapter 3, and it is elucidated why the JPDDs are considered to alleviate the comprehension problems of the textual implementations of such examples.

6.1.1 The Sanitizing Aspect Implemented With AspectJ

The first example deals with the complex join point selection of the sanitizing aspect which has been presented in section 3.1.1. The idea of the sanitizing aspect (cf. [*Masuhara & Kawauchi (2003)*]) is to sanitize any unsafe data coming from the untrusted internet before it is printed out to a dynamically generated web page. The goal is to prevent cross-site-scripting where an attacker injects a malicious script to a dynamically generated web page in order to reveal confidential data as soon as the web page is displayed on the client machine. The concrete instance of the sanitizing aspect presented in Chapter 3 of this thesis deals with the sanitation of cookie values which are retrieved from a HTTP request coming from the internet and which are supposed to be printed out to the internet again as part of an HTTP response using a "PrintWriter". The example is implemented using AspectJ.

6.1.1.1 Using JPDDs to Represent the Join Point Selection

Before representing a join point selection in terms of JPDDs, the underlying conceptual view on program execution of the join point selection needs to be determined. In other words, it needs to be decided which notational means of JPDDs (i.e. interaction diagram-based, activity diagram-based, or state chart-based means) shall be used to visualize the join point selection. Considering that the major objective of join point selection considered here is to select system events that involve objects resulting from a particular origin, the join point selection is accounted to refer to a data flow-oriented view on program execution. Accordingly, an activity diagram-based JPDD is chosen to visualize the join point selection

constraints as it is considered most capable of highlighting join point selection constraints on data flow (cf. section 5.7.2).

Correspondingly, Figure 6.1 shows an activity diagram-based JPDD which outlines the join point selection of the sanitizing aspect: the JPDD selects all invocations of methods whose name starts with "print", which are addressed to an instance of class "PrintWriter", and which take an instance of class "String" as argument (see action node identified with "?jp"). Together with the join points, the JPDD exposes the "String" argument of the intercepted "print.*" methods, so that it can be sanitized by the join point adaptation (see object node identified with "?val"). As an additional constraint, the JPDD requires that the "String" arguments of the "print.*" methods must originate from (i.e. be returned by) an invocation of a method named "getValue" which is addressed to an instance of class "Cookie". In turn, that instance of class "Cookie" must originate from (i.e. be returned by) an invocation of a method with name "getCookie" which is addressed to an instance of class "HttpServletRequest" and which takes an instance of class "String" as argument (this instance is the identifying name of the cookie). Note how the JPDD makes use of indirect control flow arrows in order to denote that the mentioned method invocations have to occur one after another, yet not necessarily *immediately* after each other.

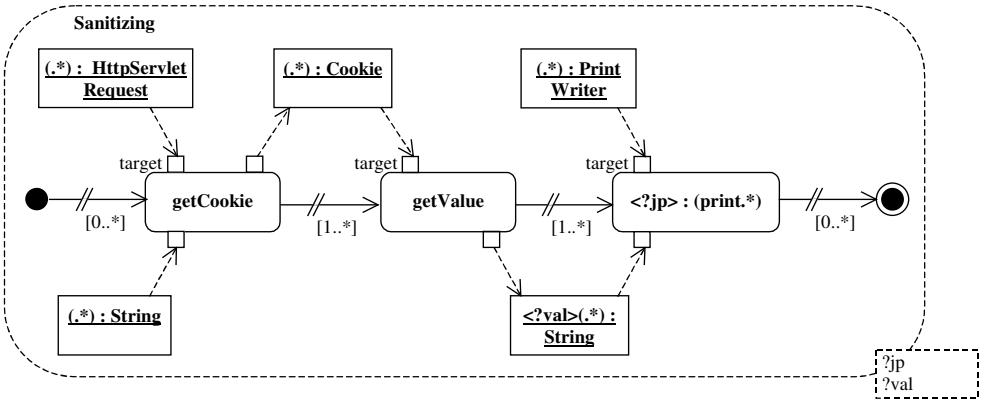


Figure 6.1 JPDD representing the join point selection of the sanitizing aspect.

6.1.1.2 Comparing the JPDD with the Pointcut Implementation in AspectJ

When comparing the JPDD shown in Figure 6.1 to the AspectJ implementation of the join point selection discussed in section 3.1.1 (see Chapter 3), one can observe that the JPDD represents data objects using distinct object symbols and that it connects these object symbols to all method symbols where they need to be involved in. In contrast to that, the AspectJ implementation defines variable names for each data object and repeats this variable name at every place in the program code where the corresponding data object needs to be involved. Furthermore, the JPDD uses an indirect control flow symbol to constrain the order of occurrence of the relevant system events, whereas the AspectJ implementation does not contain such explicit notational construct. In the AspectJ implementation, the order arises implicitly from the shared access to a common data structure. The JPDD does not define and maintain any such data structure, which is inevitable in the AspectJ implementation in order to realize the desired join point selection.

Hence, while the JPDD explicitly highlights all relevant selection constraints of the join point selections with the help of distinct notational constructs, the AspectJ implementation

tends to encrypt these constraints in the details of the pointcut implementation. As a result, readers of the AspectJ implementation are obliged to carefully search for them "manually".

Listing 6.1 Simple indication of data dependencies with help of variable names in AspectJ.

```
1  public aspect Sanitizing {
2
3    private static Collection cookies = new HashSet();
4    private static Collection values = new HashSet();
5
6    private pointcut dataOrigin():
7      call(Cookie HttpServletRequest.getCookie(String));
8
9    after() returning (Cookie cookie) : dataOrigin() {
10     cookies.add(cookie);
11   }
12
13   private pointcut dataAccess(Cookie cookie):
14     call(String Cookie.getValue()) && target(cookie)
15     && if(cookies.contains(cookie));
16
17   after(Cookie cookie) returning (String value) : dataAccess(cookie) {
18     values.add(value);
19   }
20
21   public pointcut dataDisposal(String value):
22     call(* PrintWriter.print*(String)) && args(value)
23     && if(values.contains(value));
24
25   void around(String value) : dataDisposal(value) {
26     //aspectual adaptation
27   }
28 }
```

In order to detect data dependencies, for example, readers of the AspectJ implementation need to search for all occurrences of a variable name in the program code, such that they may properly assess in which method a data object is involved in, and how (see Listing 6.1). Using the same variable name to refer to the same data object in different components of the pointcut implementation may be a hint to these data dependencies. Since the variable names are defined in different scopes, though, they are semantically independent from each other and there is no guarantee that they really refer to the same data objects. Hence, in order to be absolutely sure, readers of the AspectJ implementation need to investigate when and how the data objects are stored and retrieved to/from the data structures of the aspect (see dashed lines in Listing 6.1).

For example, readers need to find out that the exposed return value "cookie" of the method call to method "getCookie" (see line 9) is stored into the "cookies" data structure (in line 10), and that this data structure is used in pointcut "dataAccess" (see line 15) in order to ensure that the target object "cookie" of the method call to method "getValue" (see line 14) is contained in that data structure. Note from Listing 6.1 the mental efforts that readers of the AspectJ implementation have to perform in order to detect this data dependency (similar efforts are required to detect the data dependency of variable "value"). In contrast to that, readers of the JPDD are pointed to these data dependencies

straightaway by means of two directed edges, which connect the method symbol "getCookie" with the object symbol "(.*) : Cookie" and the object symbol "(.*) : Cookie" with the method symbol "getValue" (see Figure 6.1), for example.

While the recurring mention of the same variable name throughout the aspect code may point the reader of the join point selection to the presence of data dependencies, no such verbalized hint may point him/her to the chronological order in which the different system events must occur in order to lead to the selection of a join point. For example, Listing 6.2 shows the pointcut definitions of the aspect where one would probably expect such verbalized hints most. However, no mention of the chronological order of the selected system events can be found in the pointcut definitions, nor in the other parts of the aspect.

Listing 6.2 No explicit mention of chronological dependency in AspectJ.

```
 6   private pointcut dataOrigin():
 7     call(Cookie HttpServletRequest.getCookie(String));
     [...]
13   private pointcut dataAccess(Cookie cookie):
14     call(String Cookie.getValue()) && target(cookie)
15     && if(cookies.contains(cookie));
     [...]
21   public pointcut dataDisposal(String value):
22     call(* PrintWriter.print*(String)) && args(value)
23     && if(values.contains(value));
```

Thus, when investigating the chronological dependencies between the different system events, readers of the AspectJ implementation may optimistically assume that the order in which the different system events have to occur in the dynamic execution of a program complies to the order in which the pointcuts and advice are specified in the aspect code. However, the order in which pointcuts and advice are specified in AspectJ code is arbitrary[44] and does not affect the overall behavior of the aspect. In consequence and in order to be absolutely sure, readers of the AspectJ implementation must carefully investigate the program code (see Listing 6.3) and must reconstruct the chronological dependencies by evaluating the data dependencies that have already been identified previously (see Listing 6.1).

For example, readers need to observe that the occurrence of a method call to method "getCookie" (see pointcut "dataOrigin" in lines 6+7 of Listing 6.3) leads to the addition of a "cookie" to the "cookies" data structure (in line 10). This data structure is then used by pointcut "dataAccess" (see lines 13-15) to define a precondition which states that a "cookie" must be contained in the "cookies" data structure (see line 15) for its "value" to be added to the "values" data structure (in line 18). The "values" data structure, in turn, is then used by pointcut "dataDisposal" (see lines 21-23) to define a precondition stating that the "value" being printed to the "PrintWriter" must be contained in the "values" data structure (see line 23) for the aspectual adaptation to take place (in line 26). Hence, after careful inspection of the program code, readers of the AspectJ implementation are able to confirm that a join point adaptation will indeed only take place if all three system events have occurred in the order in which they are specified in the aspect code. In order to come to

[44] at least in this case since all pointcuts refer to distinct join points (i.e. to method calls of different methods).

that conclusion, readers of the AspectJ code had to attentively scrutinize all semantic implications of the program code (i.e. they basically had to "execute" the program code "in their minds"). In contrast to that, the JPDD explicates the chronological dependency between the respective system events by means of two indirect relationships which connect method symbol "getCookie" with method symbol "getValue" and method symbol "getValue" with method symbol "print" (see Figure 6.1), which should substantially facilitate the detection of this chronological dependency.

Listing 6.3 Detection of chronological dependencies by evaluating data dependencies in AspectJ.

```
1  public aspect Sanitizing {
2
3    private static Collection cookies = new HashSet();
4    private static Collection values = new HashSet();
5
6    private pointcut dataOrigin():
7      call(Cookie HttpServletRequest.getCookie(String));
8
9    after() returning (Cookie cookie) : dataOrigin() {
10     cookies.add(cookie);
11   }
12
13   private pointcut dataAccess(Cookie cookie):
14     call(String Cookie.getValue()) && target(cookie)
15     && if(cookies.contains(cookie));
16
17   after(Cookie cookie) returning (String value) : dataAccess(cookie) {
18     values.add(value);
19   }
20
21   public pointcut dataDisposal(String value):
22     call(* PrintWriter.print*(String)) && args(value)
23     && if(values.contains(value));
24
25   void around(String value) : dataDisposal(value) {
26     //aspectual adaptation
27   }
28 }
```

6.1.1.3 Summary

In summary, the JPDD representation is considered to improve over the AspectJ implementation of the join point selection insofar that it represents each system event and each data object by a single and distinct entity, and that it explicitly highlights all data dependencies and all chronological dependencies that must exist between them by distinct data flow and control flow arrows. Thus, readers of the join point selection are freed from carefully searching for these dependencies in the details of the pointcut implementation. That is, readers do not need to investigate multiple accesses to a common data structure. And they do not need to contemplate on their implications, i.e. on both the data dependencies and the chronological dependencies that might arise from these accesses. In consequence, it should be much easier for the reader of a JPDD – as compared to the reader of the AspectJ implementation – to identify the key selection constraints that must be fulfilled by a runtime situation in order to be selected by the join point selection.

The activity diagram-based notation chosen here to represent the join point selection is considered to significantly contribute to that easier comprehension because it reflects on the conceptual view on program execution which underlies the join point selection: a major concern of the join point selection considered here is to select method calls which involve objects that result from a particular other method call. Activity diagram-based JPDDs are capable of rendering these objects using distinct symbols and of highlighting all of the relationships in which these objects must be involved. As a result, readers of the join point selection may detect the importance of this data flow to the join point selection more easily than in a representation where objects are mere "annotations" of method invocations and where the identification of "same" objects must be accomplished in a purely intellectual manner (such as in case of the AspectJ implementation).

6.1.2 The Safe Iterator Aspect Implemented With AspectC++

The next example deals with the safe iterator aspect which has been presented in section 3.1.2. The objective of the safe iterator aspect (cf. [*Allan et al. (2005)*]) is to react appropriately on the inappropriate use of invalidated iterators, i.e. of iterators that have been created for data structures which have been modified in the meanwhile. The concrete safe iterator aspect which has been presented in Chapter 3 of this thesis aims to safeguard the usage of iterators on char vectors in C++.

6.1.2.1 Using JPDDs to Represent the Join Point Selection

In order to visualize the join point selection of the safe iterator aspect in an appropriate way, the underlying conceptual view on program execution of the join point selection needs to be determined at first: the objective of the join point selection is to catch hold on a sequence of system events where two objects (i.e. the iterator and its underlying data structure) are recurrently involved in multiple successive actions (i.e. in the creation and the usage of the iterator, and in the update of the data structure). Accordingly, the conceptual view on program execution which underlies the join point selection of the safe iterator aspect can be accounted to be a data flow-oriented view. And thus, an activity diagram-based JPDD is used to represent the join point selection of the aspect.

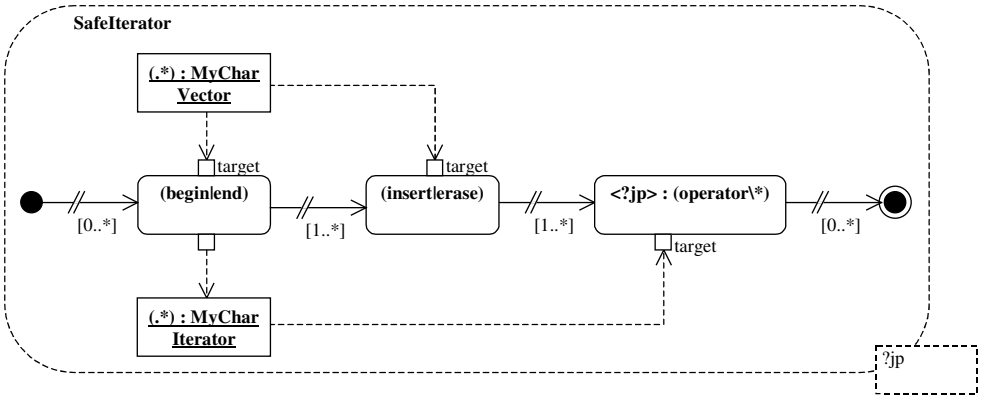


Figure 6.2 JPDD representing the join point selection of the safe iterator aspect.

Correspondingly, Figure 6.2 shows an activity diagram-based JPDD which specifies the join point selection of the safe iterator aspect. The JPDD outlines all scenarios in which the usage of "MyCharIterators" on "MyCharVectors" is unsafe in C++. As join points, the

JPDD selects all usages of the dereference operator "*" that are issued on instances of "MyCharIterator" (see action node identified with "?jp"). As an additional constraint, the dereferenced instance of "MyCharIterator" must have been created from an instance of "MyCharVector" using its methods "begin" or "end" (i.e. using a method with name "begin" or with name "end"). Finally, that very instance of "MyCharVector" must be involved in (i.e. must be target of) an invocation of a method with name "insert" or "erase" (leading to the update of the vector). The JPDDs specifies that the invocation of the update methods (i.e. of a method with name "insert" or "erase") must occur between the creation of the iterator (using a method with name "begin" or "end") and the dereference of the iterator (using the dereference operator "*"). Note how the JPDD makes use of indirect control flow arrows in order to denote that the mentioned method invocations do not have to occur right after each other.

6.1.2.2 Comparing the JPDD with the Pointcut Implementation in AspectC++

When comparing the JPDD shown in Figure 6.2 to the AspectC++ implementation of the join point selection discussed in section 3.1.2 (Chapter 3), one can make similar observations as in the previous section (where a JPDD was compared to an equivalent AspectJ implementation): while the JPDD renders all data objects with the help of distinct object symbols, the AspectC++ implementation makes use of variable names. And while the JPDD connects the object symbols to the symbols of all methods where the data objects need to be involved in, the AspectC++ implementation repeats the variable names of those data objects at every place in the program code where the corresponding object is involved in. The chronological order of the method invocations is explicitly highlighted in the JPDD using indirect relationships, whereas no such explicit language construct is present in the program code. Finally, the AspectC++ implementation makes extensive use of data structures in order to enforce the required dependencies. Furthermore, additional pointcuts and advice are needed to cope with context switches, variable assignments, and variable deallocations, which are necessary in order to keep track of all objects that need to be affected by the aspect. In contrast to that, the JPDD abstracts over such details.

Considering that the differences between the AspectC++ implementation and the JPDD of the join point selection are so similar to the differences between the AspectJ implementation and the JPDD discussed in the previous section, the problems of the program code and thus the benefits of JPDDs are comparable, too:

The use of same variable names to denote same data objects, for example, may only suggest the existence of data dependencies because the variable names are defined in different scopes and thus are semantically independent from each other. In consequence, readers of the AspectC++ implementation must carefully investigate the maintenance of the data structures in order to be absolutely sure about these data dependencies. Listing 6.4 illustrates the intellectual efforts that readers of the AspectC++ implementation need to perform in order to assert that same variable names actually refer to same data objects. In lines 24-26, for example, readers need to observe that the return value (`tjp->result()`) of the creation of an iterator is used as a key value ("it") in order store data values into the data maps "it_vec" and "it_vec_state". Subsequently, readers need to discover that the same data maps are used in lines 34+35 again. This time, the executing instance (`that(it)`) of the dereference operation is used as a key value ("it") in order to retrieve the data values again. As a result, readers may conclude that both values "it" must refer to the same objects for the evaluation in lines 34+35 to make sense and to be

accomplishable, and thus for the join point adaptation to (possibly) apply. In contrast to that, the JPDD shown in Figure 6.2 highlights this dependency by means of a single object symbol and by two relationships which connect the corresponding object symbol with the appropriate method symbols. This is considered to be for the sake of the reader of the join point selection who is able to observe this dependency right away now.

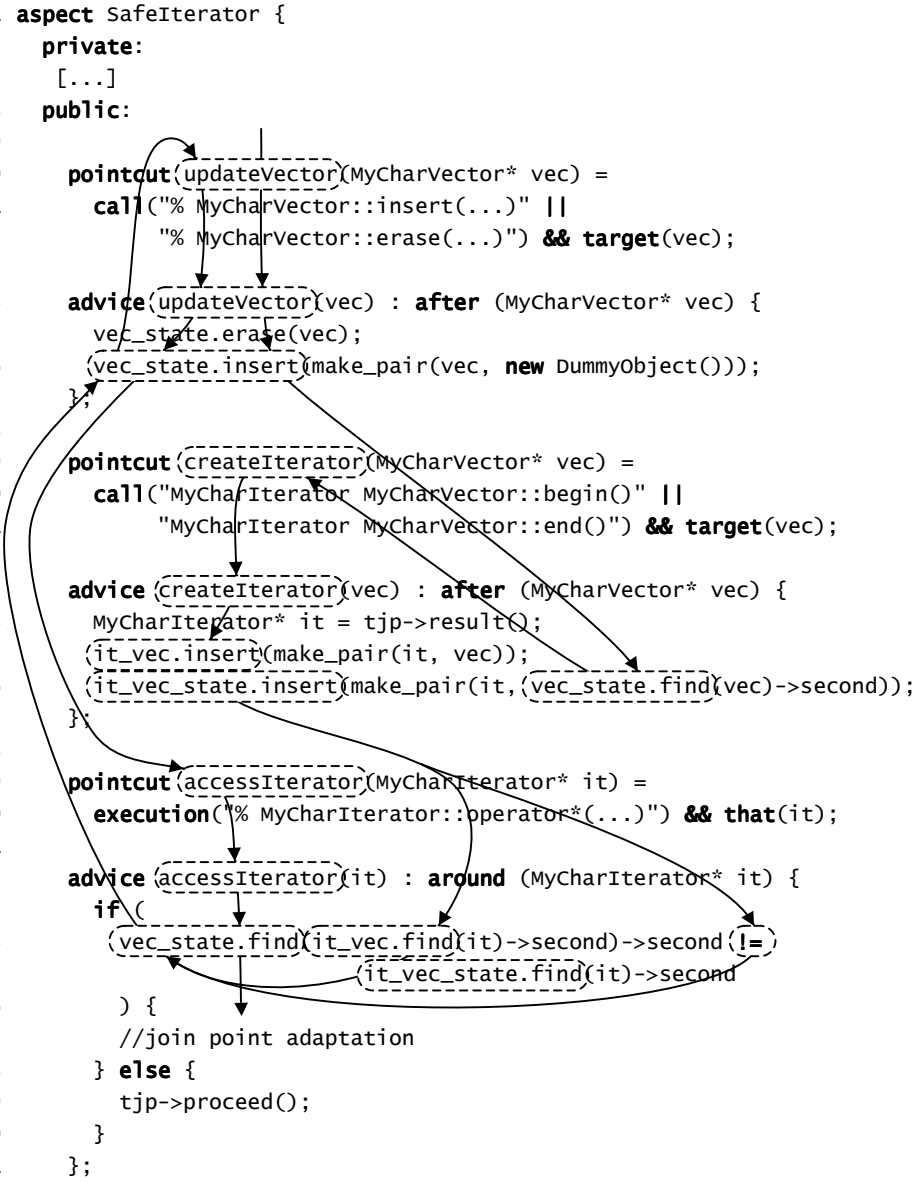
Listing 6.4 Simple indication of data dependencies with help of variable names in AspectC++.

```
1  aspect SafeIterator {
2    private:
3      map<MyCharVector*,DummyObject*> vec_state;
4      map<MyCharIterator*,MyCharVector*> it_vec;
5      map<MyCharIterator*,DummyObject*> it_vec_state;
6      typedef map<MyCharIterator*,MyCharVector*>::iterator MapIterator;
7
8    public:
9
10     pointcut updateVector(MyCharVector* vec) =
11       call("% MyCharVector::insert(...)" ||
12            "% MyCharVector::erase(...)") && target(vec);
13
14     advice updateVector(vec) : after (MyCharVector* vec) {
15       vec_state.erase(vec);
16       vec_state.insert(make_pair(vec, new DummyObject()));
17     };
18
19     pointcut createIterator(MyCharVector* vec) =
20       call("MyCharIterator MyCharVector::begin()" ||
21            "MyCharIterator MyCharVector::end()") && target(vec);
22
23     advice createIterator(vec) : after (MyCharVector* vec) {
24       MyCharIterator* it = tjp->result();
25       it_vec.insert(make_pair(it, vec));
26       it_vec_state.insert(make_pair(it, vec_state.find(vec)->second));
27     };
28
29     pointcut accessIterator(MyCharIterator* it) =
30       execution("% MyCharIterator::operator*(...)") && that(it);
31
32     advice accessIterator(it) : around (MyCharIterator* it) {
33       if (
34         vec_state.find(it_vec.find(it)->second)->second !=
35                        it_vec_state.find(it)->second
36         ) {
37         //join point adaptation
38       } else {
39         tjp->proceed();
40       }
41     };
```

Likewise, readers of the AspectC++ implementation need to manually deduce the chronological dependencies between the relevant system events of the join point selection because there is no programmatic language construct in the program code which would externalize these dependencies. Thus, readers of the AspectC++ implementation need to

carefully reconstruct the order in which these system events need to occur in the dynamic execution of a program by evaluating the maintenance operations on a common data structure. Similar to the AspectJ implementation discussed in the previous section, the order in which the pointcuts and advice are defined in the aspect may give a hint to the order in which these system events need to occur in the dynamic execution of a program. However, this hint is rather vague because the specification order of pointcuts and advice does not affect the overall behavior of the AspectC++ aspect[45]. Hence, readers of the AspectC++ implementation need to reconstruct those chronological dependencies from the data dependencies which they have already identified. Listing 6.5 indicates that this task is a bit more intricate than in case of the AspectJ implementation considered in the previous section.

Listing 6.5 Detection of chronological dependencies by evaluating data dependencies in AspectC++.



```
1  aspect SafeIterator {
2    private:
       [...]
8    public:
9
10     pointcut updateVector(MyCharVector* vec) =
11       call("% MyCharVector::insert(...)" ||
12            "% MyCharVector::erase(...)") && target(vec);
13
14     advice updateVector(vec) : after (MyCharVector* vec) {
15       vec_state.erase(vec);
16       vec_state.insert(make_pair(vec, new DummyObject()));
17     };
18
19     pointcut createIterator(MyCharVector* vec) =
20       call("MyCharIterator MyCharVector::begin()" ||
21            "MyCharIterator MyCharVector::end()") && target(vec);
22
23     advice createIterator(vec) : after (MyCharVector* vec) {
24       MyCharIterator* it = tjp->result();
25       it_vec.insert(make_pair(it, vec));
26       it_vec_state.insert(make_pair(it, vec_state.find(vec)->second));
27     };
28
29     pointcut accessIterator(MyCharIterator* it) =
30       execution("% MyCharIterator::operator*(...)") && that(it);
31
32     advice accessIterator(it) : around (MyCharIterator* it) {
33       if (
34         vec_state.find(it_vec.find(it)->second)->second !=
35                                    it_vec_state.find(it)->second
36         ) {
37         //join point adaptation
38       } else {
39         tjp->proceed();
40       }
41     };
```

[45] at least in this case since all pointcuts refer to distinct join points (i.e. to method calls of different methods).

For example, readers of the AspectC++ implementation need to observe that every update of a vector (intercepted by pointcut "updateVector"; see line 10 in Listing 6.5) leads to an update of the data map "vec_state" (in line 16), which is used to remember the state of a vector in data map "it_vec_state" (in line 26) whenever an iterator is created (the creation of an iterator is intercepted by pointcut "createIterator"; see line 19). Data map "it_vec_state" is then used (in lines 34+35) to compare the state of a vector (stored in data map "vec_state") when the iterator is used (intercepted by pointcut "accessIterator") with the state of a vector at the time when the iterator was created (stored in data map "it_vec_state"). The join point adaptation shall only apply if the respective states are *not* equal (see "!=" operator in line 34), which means that a new update of the data map "vec_state" must have occurred (and intercepted and taken account of by pointcut and advice "updateVector" in lines 10-17) in the meanwhile, i.e. since the iterator has been created. Only if this has been the case, i.e. if the current state and the remembered state are *not* equal, the dereference operation of the iterator (intercepted by pointcut "accessIterator"; see line 29) leads to the join point adaptation (in line 37). In summary, readers of the AspectC++ implementation need to unravel the intricate data interdependencies between the system event in order to unravel a complex and interlaced chronological order between them. In contrast to that, the JPDD clearly highlights the relevant steps of that order with the help of indirect control flow symbols, which should make the detection of these dependencies much easier.

Listing 6.6 Copoing with context switches, variable assignments and deallocations in AspectC++.

```
43    pointcut copyIterator(MyCharIterator* lhs) =
44    construction("MyCharIterator") && args("const MyCharIterator&") && that(lhs);
      [...]
56    pointcut assignIterator(MyCharIterator* lhs) =
57      call("% MyCharIterator::operator=(const MyCharIterator&)") && target(lhs);
      [...]
70    pointcut destructIterator(MyCharIterator* it) =
71      destruction("MyCharIterator") && that(it);
      [...]
78    pointcut destructVector(MyCharVector* vec) =
79      destruction("MyCharVector") && that(vec);
```

Other than in AspectJ, readers of the AspectC++ implementation are confronted with even more interdependencies which result from the presence of pointcuts and advice (see Listing 6.6) that deal with the bare storage management of AspectC++ (i.e. variable assignments, context switches, and variable deallocations). These pointcuts and advice ensure that the aspect does not loose track of iterators when they are duplicated or when they are passed on to/returned from other methods, and they make sure that iterators are removed from the data structures of the aspect whenever they are deallocated. Readers of the AspectC++ implementation need to find out that the occurrence and execution of the former two pointcuts and advice (named "copyIterator" and "assignIterator") is facultative, while the occurrence and execution of the latter two pointcuts and advice (named "destructIterator" and "destructVector") is prohibitive for the join point selection to apply (note that these interdependencies are not visualized in Listing 6.6). In contrast to that, JPDDs abstract over such details and thus free the readers of the join point selection from contemplating such issues.

6.1.2.3 Summary

In summary, the JPDD shown in Figure 6.2 improves over the AspectC++ implementation of the join point selection because it frees readers from the need to scrutinize the semantic implications of the various components of the pointcut implementation in order to reconstruct data dependencies and chronological dependencies that arise between relevant system events. The readers do not need to cope with any complex data structures, nor with their interfaces. Likewise, they are not confronted with the complexity of tracing relevant objects when they are passed around from one method scope to another, or when they are assigned to a new variable, etc. As a result, readers of the join point selection are expected to identify and comprehend the key selection criteria of the join point selection much easier when they study a join point selection which is represented as a JPDD. This is because, in contrast to the AspectC++ implementation, JPDDs externalize these data dependencies and chronological dependencies with the help of distinct relationship symbols. Furthermore, a JPDD focuses only on system events which are key to the objectives of the join point selection.

The activity diagram-based notation used to represent the join point selection is expected to be particularly helpful in that regard because it emphasizes the underlying conceptual view of the join point selection. That is, it stresses that the same objects must be involved in different system events for the join point selection to select a join point. It does so by rendering each of these objects using a distinct symbol and by highlighting the involvement of these objects in the corresponding method calls using pictured relationships. This is expected to facilitate the detection of the data dependencies between the system events, in particular in comparison to a representation where data dependencies need to be realized using shared accesses on a common data structure (such as it is the case in the AspectC++ implementation of the join point selection).

6.1.3 The File Access Management Aspect Implemented With AspectCOBOL

The following example deals with the join point selection of the file access management aspect which has been presented in section 3.1.3. The goal of the file access management aspect (cf. [*Lämmel & Schutter (2005)*]) is to intercept unsafe file accesses to closed files in order to open the file first and then resume with the intercepted behavior. The concrete file access management aspect presented in Chapter 3 of this thesis deals with unsafe file accesses implemented in COBOL code. Accordingly, the file access management aspect is implemented in AspectCOBOL.

6.1.3.1 Using JPDDs to Represent the Join Point Selection

Before the join point selection of the file access management aspect may be represented using a JPDD, its underlying conceptual view on program execution needs to be determined first. Considering that the key objective of the join point selection of the file access management aspect is to intercept operations on objects (i.e. files) which are in a particular state (i.e. in state "closed") – and that the aspect discontinues interception when objects leave that particular state, and that it resumes interception as soon as the objects transition to that particular state again – the join point selection can be accounted to refer to a state-oriented view on program execution. Accordingly, a state chart-based JPDD shall be used to represent the join point selection.

Figure 6.3 outlines how such a state chart-based representation of the join point selection of the file access management aspect looks like using a JPDD. As join points, the JPDD selects all method invocations of methods which are named "read", "rewrite", "write", "delete", or "start" (see state transition identified with "?jp"). The join point selection takes place only, however, as long as no method with name "open" has been called. Once this happens, the join point selection transitions to another state and stops selecting any further method call just until a method with name "close" is invoked. After the invocation of a method with name "close", the selections of the aforementioned method calls resumes – just until the next invocation of a method with name "open" (and so forth). Note that all mentioned method invocations must target the same object, i.e. an instance of "File" (as indicated by the enclosing object symbol labeled with "(.*) : File"). Note furthermore that the JPDD assigns identifiers to the states of the object (i.e. "?closed" and "?open") just to suggest to the reader in what object state the join point selection takes place, and in what object state it does not.

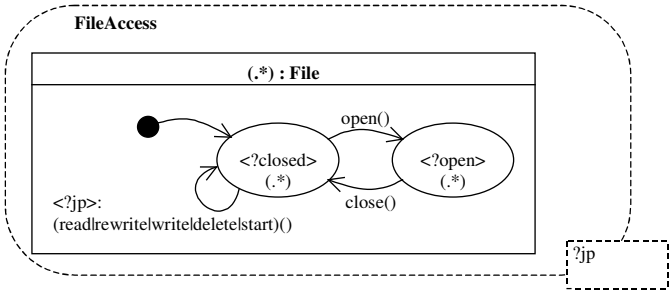


Figure 6.3 JPDD representing the join point selection of the file access management aspect.

6.1.3.2 Comparing the JPDD with the Pointcut Implementation in AspectCOBOL

When comparing the JPDD shown in Figure 6.3 to the AspectCOBOL implementation discussed in section 3.1.3 (Chapter 3), a major difference between the two specifications of the join point selection is the representation of states and state transitions: while the JPDD makes use of two distinct ellipses to represent the "open" and "closed" states of a file, the AspectCOBOL implementation makes use of a variable (i.e. `CHECKOPEN-STATE`) to do so (i.e. the value of that variable may be either `CHECKOPEN-OPEN` or `CHECKOPEN-CLOSED`). And while the JPDD makes use of arrows to denote possible state transitions, the AspectCOBOL implementation evaluates and modifies the state variable at appropriate places in the program code. Finally, the JPDD indicates the initial state of a file using a special state symbol, whereas the AspectCOBOL implementation initializes the state variable with an appropriate value (i.e. with `0` in this case, which equates to `CHECKOPEN-CLOSED`). Another difference pertains to the representation of files: while the JPDD makes use of an object symbol to represent a single file and its states, the AspectCOBOL implementation defines a dynamic table which is used to store and retrieve all file objects that need to be monitored together with their states.

In consequence, readers of the AspectCOBOL implementation need to carefully inspect the program code of the file access management aspect in order to find out which method invocations lead to which state transitions. For example, they need to find out that the occurrence of an `OPEN` call (see line 24 in Listing 6.7) makes the corresponding file go to state `CHECKOPEN-OPEN` (see line 36), whereas the occurrence of a `CLOSE` call (see

line 39) makes the file go to state CHECKOPEN-CLOSED (see line 48). Apart from that, they need to find out under which circumstances (i.e. in which states) state transitions may be triggered at all. For example, readers of the AspectCOBOL implementation need to find out that the occurrence of a file access (i.e. of a READ, REWRITE, WRITE, DELETE, or START call; see lines 52+53) – which triggers the join point adaptation (in line 62) – may only occur if the corresponding file is in state CHECKOPEN-CLOSED (or if it has not been added to the dynamic table of all known files yet, which denotes the same state; see line 61 in Listing 6.7). Note how, in contrast to that, the JPDD representation of states and state transitions using ellipses and arrows emphasizes the source and target states of state transitions, as well as the method calls which trigger them, and thus should make the detection of state changes which are key to the join point selection easier.

Listing 6.7 Defining states and state transitions in AspectCOBOL.

```
 7   01 DYNAMIC-TABLE.
      [...]
12      05 CHECKOPEN-STATE PIC 9   VALUE 0.
13         88 CHECKOPEN-OPEN       VALUE 1.
14         88 CHECKOPEN-CLOSED     VALUE 0.
      [...]
23    MY-OPEN-CONCERN.
24      USE BEFORE OPEN
25      [...]
26    MY-OPEN-ADVICE.
        [...]
36      SET CHECKOPEN-OPEN (CHECKOPEN-IDX) TO TRUE.
37
38    MY-CLOSE-CONCERN.
39      USE BEFORE CLOSE
40      [...]
41    MY-CLOSE-ADVICE.
        [...]
48        SET CHECKOPEN-CLOSED (CHECKOPEN-IDX) TO TRUE
49      [...]
50
51    MY-ACCESS-CONCERN.
52      USE BEFORE
53      (READ OR REWRITE OR WRITE OR DELETE OR START)
54      [...]
55    MY-ACCESS-ADVICE.
        [...]
61      IF AT-END-OF-TABLE OR CHECKOPEN-CLOSED (CHECKOPEN-IDX)
62 *      PERFORM JOIN-POINT-ADAPTATION
63      END-IF.
```

In order to recognize that state transitions shall be monitored for each file individually, readers of the AspectCOBOL implementation need to understand the definition of the complex data structure used to stored them, which is a dynamic table in this case (see lines 7-14 in Listing 6.8). This means that they need to get familiar with the concept of "levels" which are used to define complex (i.e. nested) data structures in COBOL. And they need to know how elements from these complex data structures may be searched and accessed. Apart from that, they need to understand the "special" way of defining and using Boolean values in COBOL.

Listing 6.8 Defining and accessing data structures in AspectCOBOL.

```
 7   01 DYNAMIC-TABLE.
 8    02 CHECKOPEN-ENTRY   OCCURS 999 TIMES
 9                         DEPENDING ON CHECKOPEN-MAX
10                         INDEXED BY CHECKOPEN-IDX.
11      05 CHECKOPEN-IDREF PIC 9(10).
12      05 CHECKOPEN-STATE PIC 9   VALUE 0.
13         88 CHECKOPEN-OPEN       VALUE 1.
14         88 CHECKOPEN-CLOSED     VALUE 0.
     [...]
38    MY-CLOSE-CONCERN.
39      [...]
40      BIND VAR-IDREF TO IDREF OF FILE.
41    MY-CLOSE-ADVICE.
42      SET CHECKOPEN-IDX TO 1.
43      SEARCH CHECKOPEN-ENTRY
44        AT END SET AT-END-OF-TABLE TO TRUE
45        WHEN VAR-IDREF = CHECKOPEN-IDREF (CHECKOPEN-IDX)
46          SET NOT-AT-END-OF-TABLE TO TRUE.
47      IF NOT-AT-END-OF-TABLE
48        SET CHECKOPEN-CLOSED (CHECKOPEN-IDX) TO TRUE
49      END-IF.
```

For example, readers of the AspectCOBOL implementation need to understand that the dynamic table in Listing 6.8 (defined at level `01`; see line 7) consists of (a maximum of) 999 `CHECKOPEN-ENTRY` entries (defined at level `02`; see lines 8-10), which in turn consist of a `CHECKOPEN-IDREF` field and a `CHECKOPEN-STATE` field (defined at level `05`; see lines 11+12). The latter of those fields denotes a Boolean variable which may hold the values `CHECKOPEN-OPEN` or `CHECKOPEN-CLOSED` (defined at the special level `88`; see lines 13+14). Furthermore, readers of the AspectCOBOL implementation need to know that the dynamic table may be accessed with the help of an index (i.e. `CHECKOPEN-IDX`; see line 10) which is used to traverse the dynamic table (see lines 42-46) and to retrieve or modify a value of the (fields of the) entry at the respective position (see lines 45+48).

Note how, in contrast to that, the JPDD represents files using a singular object symbol which encapsulates all states and state transitions in order to denote that they need to be monitored for each file individually. Hence, JPDDs abstract over the details of keeping track of file objects and thus free the reader of the join point selection to ponder on this issue.

6.1.3.3 Summary

In summary, the JPDD shown in Figure 6.3 is considered to improve over the AspectCOBOL implementation of the join point selection as it explicitly highlights the significance of the relevant system events to the join point selection. That is, it highlights the impact of an invocation of method "open" (which makes the join point selection to pause) as well as the impact of a – subsequent – invocation of method "close" (which makes the join point selection to resume). Readers of the JPDD do not have to search for the corresponding variable assignments in the program code in order to find out about these impacts. Furthermore, they do not need to analyze manifold accesses to a common data structure in order to recognize that the effects of these method invocations shall be

considered for each file individually, nor to get familiar with the language-specific way to define and use such common data structures. As a result, it is expected that it is much easier to identify the key selection constraints of the join point selection, and the risk of miscomprehending the overall objectives of the join point selection is much reduced.

The adoption of the ideas and concepts of state charts is considered to significantly contribute to this easier comprehension of the join point selection as it reflects on the underlying conceptual view on program execution of the join point selection. That is, the state chart-based representation of the join point selection focuses on states and state transitions rather than on variables and procedure calls (such as it is the case in the AspectCOBOL implementation of the join point selection). Thus, the particular relevance of the state to the join point selection should be easier to observe. Furthermore, it should be easier to observe when a file may transition from one state to another state, i.e. what transitions are active in what states.

6.1.4 The Contextual Logging Aspect Implemented With Alpha

The next example is about the complex join point selection which is required to implement the contextual logging aspect presented in section 3.1.4. The objective of the contextual logging aspect (cf. [*Allan et al. (2005)*]) is to log queries of users of a public information terminal only if they are logged in; queries of anonymous users should not be logged. In Chapter 3, the aspect is implemented using the aspect-oriented programming language Alpha, and it is applied to a public information terminal which is implemented in the object-oriented programming language L2.

6.1.4.1 Using JPDDs to Represent the Join Point Selection

In order to visualize the join point selection of the contextual logging aspect in an appropriate way, the underlying conceptual view on program execution of the join point selection needs to be determined first: just like in the example considered in the previous section, the contextual logging aspect affects system behavior (i.e. the execution of queries) only when the system is in a particular state (i.e. when users are logged in); if the system leaves that particular state, the aspect stops taking affect just until the system transitions to that particular state again. Accordingly, the join point selection can be accounted to realize a state-oriented view on program execution, and thus it should be represented as a state chart-based JPDD.

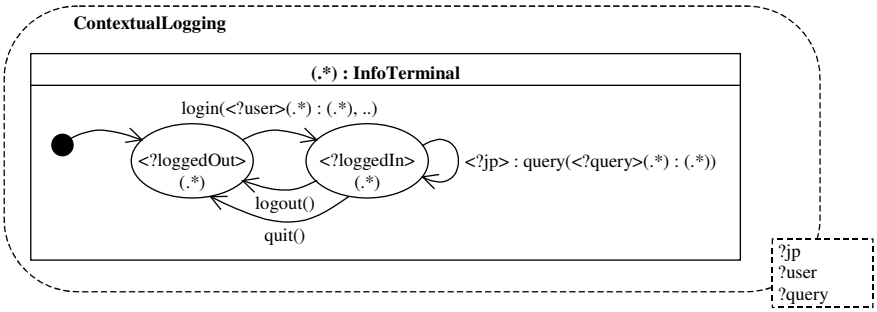


Figure 6.4 A JPDD representing the join point selection of the contextual logging aspect.

Correspondingly, Figure 6.4 represents the join point selection of the contextual logging aspect using a state chart-based JPDD. As join points, the JPDD selects all method invocations of methods whose name is "query" and which take (exactly) one argument (of

any type; ".*"). Together with the method invocations (identified with "?jp"), the JPDD exposes the arguments of the method invocations (identified with "?query") so that it can be logged by the join point adaptation. Note that the JPDD selects the mentioned method invocations only if they occur (some time) after a method call to some other method with name "login" which takes an arbitrary number of arguments of any type (at least one, though). The first argument of that method call (identified with "?user") is exposed together with the selected method call ("?jp") and its argument ("?query") to the join point adaptation, so that the query being logged can be affiliated with the user who is issuing it. Furthermore, the JPDD specifies that the join point selection stops as soon as a method invocation to a method with name "logout" or "quit" occurs. The join point selection may resume as soon as method "login" is called again (and so forth). Note that all mentioned method invocations must target the same object, i.e. an instance of "InfoTerminal". Note furthermore that the JPDD assigns identifiers to the states of the object (i.e. "?loggedOut" and "?loggedIn") merely to emphasize the meaning of such states.

6.1.4.2 Comparing the JPDD with the Pointcut Implementation in Alpha

When comparing the JPDD shown in Figure 6.4 to the Alpha implementation discussed in section 3.1.4 (see Chapter 3), one can observe that the Alpha implementation represents states and state transitions in terms of logic predicates, whereas the JPDD makes use of ellipses and arrows. In the logic predicates, the Alpha implementation makes use of several variable names in order to refer to various system events and to define their properties and their relationships, i.e. their chronological order in particular. Alternative chronological orders are defined using alternative rules. In contrast to that, the JPDD represents distinct system events using distinct transition symbols and expresses possible chronological orders of those system events by appropriately connecting their transition symbols to the state symbols. Finally, the Alpha implementation defines one further variable name in order to express a data dependency that must hold for all system events. The JPDD represents this data dependency by enclosing all states and state transitions in a single object symbol (labeled with "(.*) : InfoTerminal").

Readers of the Alpha implementation need to inspect the program code carefully in order to discover all occurrences of a variable name which refers to the same system event, such that they may reconstruct the chronological dependencies between the different relevant system events. For example, from the specification shown in Listing 6.9, they need to discover that the most recent call to method "logout" or "quit" (represented by variable `O`; see line 25) must have occurred `before` (see predicate in line 26) the most recent call to method "login" (represented by variable `I`; see line 22), which – in turn – must have occurred `before` (see predicate in line 24) the current call to method "query" (represented by variable `Q`; see line 20) in order to satisfy the join point selection criteria. Alternatively, as specified in Listing 6.10, the call to method "logout" or "quit" must *not* have occurred so far (see lines 15+16) – provided that, still, a method call to method "login" has occurred prior to the current method call to method "query" (see lines 13+14).

Another comprehension problem may arise from the fact that the chronological order of system events is specified with respect to the execution trace of a program. That is, the rules describe two alternative scenarios in which logging *shall* occur. No explicit statement is made describing possible scenarios in which logging shall *not* occur. As a result, the interrupting nature of a call to methods "logout" or "quit" which makes the aspect stop logging queries just until method "login" is called again is difficult to detect.

Listing 6.9 Detecting chronological dependencies in Alpha.

```
19 methodsToLog(T,USER,QUERY) :-
20   calls(Q,_,T,query,QUERY),
21   now(Q),
22   mostRecent(I,calls(I,_,T,login,_)),
23   calls(I,_,_,_,USER),
24   before(I,Q),
25   mostRecent(O,or(calls(O,_,T,logout,_),calls(O,_,T,quit,_))),
26   before(O,I),
27   classof(T,infoterminal).
```

Listing 6.10 Selecting query calls after initial user login with the help of "negation as failure" in Alpha.

```
10 methodsToLog(T,USER,QUERY) :-
11   calls(Q,_,T,query,QUERY),
12   now(Q),
13   calls(I,_,T,login,USER),
14   before(I,Q),
15   \+calls(_,_,T,logout,_),
16   \+calls(_,_,T,quit,_),
17   classof(T,infoterminal).
```

In contrast to that, the JPDD renders two distinct states in order to explicitly distinguish between scenarios in which logging shall occur (see state "?loggedIn") and scenarios in which logging shall not occur (see state "?loggedOut"). In consequence, readers of the JPDD may recognize more easily that logging shall only occur while users are "logged in"[46]. Apart from that, readers of the JPDD do not need to map variable names in order to identify the (chronological) dependencies between the relevant system events. Nor to they need to get familiar with the fundamental basics of PROLOG (such as with the "negation as failure" operator used in lines 15+16 of Listing 6.10) in order to understand that the occurrence of a call to method "logout" or "quit" is no categorical requirement for the join point selection to take effect.

Listing 6.11 Detecting data dependencies in Alpha.

```
19 methodsToLog(T,USER,QUERY) :-
20   calls(Q,_,T,query,QUERY),
21   now(Q),
22   mostRecent(I,calls(I,_,T,login,_)),
23   calls(I,_,_,_,USER),
24   before(I,Q),
25   mostRecent(O,or(calls(O,_,T,logout,_),calls(O,_,T,quit,_))),
26   before(O,I),
27   classof(T,infoterminal).
```

Other than readers of the JPDD, readers of the Alpha implementation are furthermore obliged to inspect all occurrences of variable names referring to the information terminal instance (represented by variable `T`) in order to reconstruct the data dependency that must

[46] Note that the Alpha implementation discussed in section 3.1.4 (Chapter 3) contains another rule which specifies that the call to method "query" must come to pass *in between* a call to method "login" and a call to the methods "logout" or "quit"; the specification of the chronological order of the method calls in this rule is facultative and thus purely declarative, though, and could be spared without altering the overall behavior of the aspect.

be satisfied between the different system events (see Listing 6.11). In contrast to that, readers of the JPDD can tell from the encapsulation of the states machine in an object symbol (labeled with "(.*) : InfoTerminal" in Figure 6.4) that all method calls must be issued on the same instance.

6.1.4.3 Summary

In summary, the JPDD shown in Figure 6.4 is considered to improve over the Alpha implementation of the join point selection because it frees readers from the burden to carefully map variable names in the many predicates of the join point selection rules, such that they may properly detect all chronological dependencies and data dependencies that must be satisfied by a runtime situation in order to denote a join point. Moreover, it frees the readers from the need to get familiar with the principles of logic programming, and with the basics of PROLOG in particular (such as with the "negation as failure" operator, for example), before they are able to properly understand the rule definitions.

Apart from that, the state chart-based representation of the join point selection shown in Figure 6.4 is considered to be particularly helpful in observing the overall objectives of the join point selection because it emphasizes the underlying conceptual view on program execution of the join point selection: the explicit representation of states helps readers of the join point selection to distinguish between runtime situations in which the contextual logging aspects is active and runtime situations in which the aspect is not active; furthermore, the transitions connecting these states give a consolidated view on all possible action/event sequences which lead to either of these runtime situations. By the help of these means, the identification of the key selection constraints and of the major objectives of the join point selection should be much facilitated – in particular in comparison to a representation where join point selections may only be specified in terms of patterns over the execution trace of a program (such as in the Alpha implementation of the join point selection).

6.2 Discussing More Examples

In addition to the examples presented in Chapter 3, this section discusses the examples which are illustrated in Appendix A of this thesis. In contrast to the previous examples, (most of) the following examples are represented with help of interaction diagram-based JPDDs (while the previous examples have been represented using activity diagram-based JPDDs or state chart-based JPDDs).

6.2.1 The Server Test Aspect Implemented With Aquarium

The first example represents a join point selection of a server test aspect, which is presented in section A.1 (see Appendix A). The server test aspect (cf. [*Nishizawa et al. (2004)*]) is supposed to verify if a request to register a user with an authentication server actually leads to a corresponding request to add the user to the database. In the Appendix A, the server test aspect is implemented with Aquarium, which is an aspect-oriented toolkit for the programming language Ruby.

6.2.1.1 Using JPDDs to Represent the Join Point Selection

In order to represent the join point selection of the server test aspect using a JPDD, it is necessary to identify the underlying conceptual view on program execution of the join point selection first. Considering that the goal of the aspect is to monitor if a particular action triggers a particular other action, which implies that the latter action needs to occur in the control flow of the former action, the underlying conceptual view on program execution can be accounted to be a control flow-oriented view. Thus, an interaction diagram-based JPDD shall be used to represent the join point selection.

Accordingly, Figure 6.5 shows an interaction diagram-based JPDD which outlines the join point selection of the server test aspect. As join points, the JPDD selects all invocations of methods which are named "addUser", which are addressed to an instance of class "DbServer", and which take (exactly) two instances of class "String" as arguments. Together with the method invocations (identified with "?jp"), the JPDD exposes both of its arguments (identified with "?usr2" and "?pw2"). The JPDD requires that the selected method invocations must come to pass in the control flow[47] of another method call, i.e. of another invocation of a method with name "registerUser" which is addressed to an instance of class "AuthServer" and which takes (exactly) two instances of class "String" as arguments. To do so, the JPDD makes use of an indirect message and indirect activation bars which are place in between the two method calls. The JPDD exposes both arguments of that earlier method invocations, too. To do so, the arguments are identified with "?usr1" and "?pw1", and the identifiers are placed in the output parameter box at the lower right corner of the JPDD.

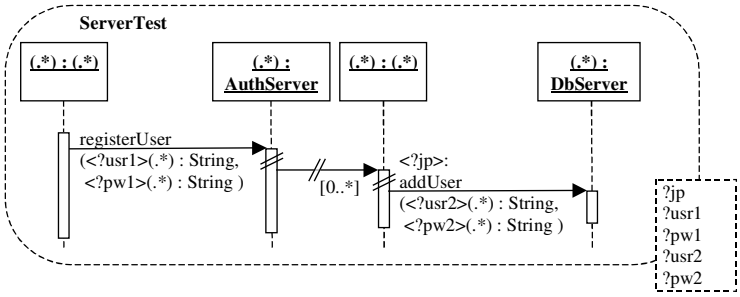


Figure 6.5 JPDD representing the join point selection of the server test aspect.

6.2.1.2 Comparing the JPDD with the Pointcut Implementation in Aquarium

When comparing the JPDD shown in Figure 6.5 to the Aquarium implementation discussed in section A.1 (see Appendix A), one can observe that the Aquarium implementation makes use of a special data structure in order to implement the control flow-dependent behavior of the aspect (see line 4 in Listing 6.12). That data structure is needed to keep track of the relevant control flow of the program (see lines 9-11), similar to what may be accomplished with the help of the `cflow` pointcut designator in AspectJ. Unfortunately, however, there is no equivalent to the `cflow` pointcut designator of AspectJ available in Aquarium and therefore the selection constraint on the control flow needs to be enforced manually.

[47] at some deeper call stack level.

As a result, readers of the pointcut implementation need to carefully investigate the program code in order to find out about that control flow dependency. That is, they need to discover that an invocation of method "registerUser" (see line 7 in Listing 6.12) leads to an increment of the data structure (in line 9). Afterwards, the program execution may continue normally (see the `proceed` call in line 10), until a method call to method "addUser" may occur (see line 14). That method call may lead to the execution of the join point adaptation (in line 23), provided that the data structure indicates that the method is called within the control flow of the "registerUser" method (see line 17). Finally, the data structure is decremented again as soon as method "registerUser" terminates (see line 11).

Listing 6.12 Detecting control flow dependencies in Aquarium.

```
 4 cflowstack = Array.new
 5
 6 Aspect.new :around, :calls_to => [:registerUser], \
 7   :on_modules_and_descendents => [:AuthServer] \
 8   do |jp, obj, *args|
 9   cflowstack.push [args[0], args[1]]
10   jp.proceed
11   cflowstack.pop
12 end
13
14 Aspect.new :around, :calls_to => [:addUser], \
15   :on_modules_and_descendents => [:DbServer] \
16   do |jp, obj, *args|
17   if cflowstack.length > 0
       [...]
23     #join point adaptation using user1, password1, user2, and password2
24   end
25   [...]
26 end
```

In conclusion, the readers of the Aquarium implementation need to carefully observe the shared accesses of the data structure in order to recognize that for the join point adaptation to take place, one method needs to occur in the control flow of the other method. While doing so, there is a risk that readers might misinterpret these shared accesses as an implementation of an ordinary data dependency. In contrast to that, the JPDD makes use of indirect messages and indirect activation bars in order to represent the control flow-dependent selection constraint of the join point selection. With help of these symbols, the JPDD aims to highlight the nesting nature of the method calls which is essential to the join point selection considered here. At the same time, it seeks to prevent readers from confusing this control-flow dependent selection constraint with a conventional data dependency.

Apart from keeping track of control flow, the data structure is also used to remember the arguments of the initial method call (i.e. of the call to method "registerUser"; see line 9 in Listing 6.13). These arguments are retrieved from the data structure upon occurrence of the second method call (i.e. the call to method "addUser"; see lines 18-20) in order to provide them to the join point adaptation (in line 23). While readers of the Aquarium implementation (once again) need to carefully investigate the program code in order to recognize where these arguments stem from (see Listing 6.13 for an illustration), the JPDD exposes the arguments with the help of two identifiers (i.e. "?usr1" and "?pw1"), which are

assigned to the parameters of the initial method call and which are exposed in the export parameter box of the JPDD. That way, the origin of the exposed parameters should be easier to detect.

Listing 6.13 Detecting origin of exposed context of control flow in Aquarium.

```
 4 cflowstack = Array.new
 5
 6 Aspect.new :around, :calls_to => [:registerUser], \
 7   :on_modules_and_descendents => [:AuthServer] \
 8   do |jp, obj, *args|
 9   cflowstack.push [args[0], args[1]]
10   jp.proceed
11   cflowstack.pop
12 end
13
14 Aspect.new :around, :calls_to => [:addUser], \
15   :on_modules_and_descendents => [:DbServer] \
16   do |jp, obj, *args|
17   [...]
18     exposeddata = cflowstack.last
19     user1 = exposeddata[0]
20     password1 = exposeddata[1]
       [...]
23     #join point adaptation using user1, password1, user2, and password2
24   end
25   [...]
26 end
```

6.2.1.3 Summary

In summary, the interaction diagram-based JPDD improves over the Aquarium implementation of the join point selection in so far that it highlights the presence of the control flow constraint in the join point selection and that it emphasizes the context exposure of the two arguments of the initial method call (i.e. of the call to method "registerUser"). Consequently, readers of the join point selection do not need to investigate any shared access to a common data structure to find out about that (and they do not need to get familiar with the interface of that data structure first). As a result, it is assumed that readers of the join point selection are prevented from overlooking the special relevance of the control flow to the join point selection as well as from misinterpreting the shared access to the common data structure as an ordinary data dependency. Hence, it is expected that readers may comprehend the key objectives of the join point selection much easier from the JPDD than from the Aquarium implementation.

The interaction diagram-based representation of the join point selection is assumed to be particularly helpful in this regard because it complies to the conceptual view on program execution which underlies the join point selection. That is, it emphasizes (by means of activation bars) that the execution of a particular method must not have terminated yet, while another method is invoked. Hence, it should be easier for readers of the join point selection to detect that the request to add a user to a database must come to pass in the control flow of a (i.e. another) request to register a user to an authentication server. This is in contrast to a notation where a variable (or data structure) is needed in order to flag if the

program execution has entered (or exited) the control flow of a particular method call (such as it is the case in the Aquarium implementation).

6.2.2 The Decorator Test Aspect Implemented With Tracematches

The next example outlines a join point selection which is required to implement a decorator test aspect, as is presented in section A.2 (see Appendix A). The decorator test aspect may be used to detect nested re-invocations of the same decorator object (which would possibly lead to an infinite loop). A sample scenario where this may happen is the combined use of the decorator pattern and the observer pattern (which is common in GUI implementations, for example). In this scenario, the decorator may illegally update the state of an observed subject (thus leading to a re-notification of the observer and a subsequent re-invocation of the decorated method, and so forth). In Appendix A, the decorator test aspect is implemented using Tracematches, and it monitors nested re-invocations of the "print(.*)" methods of "OutputDecorator" objects.

6.2.2.1 Using JPDDs to Represent the Join Point Selection

In order to visualize the join point selection of the decorator test aspect in an appropriate way using JPDDs, the underlying conceptual view on program execution of the join point selection needs to be determined first: the objective of the join point selection is to intercept nested actions. Accordingly, the join point selection can be accounted to refer to a control flow-oriented view on program execution. Therefore, an interaction diagram-based JPDD (see Figure 6.6) is used to represent the join point selection – similar to the example presented in the previous section – as this representation has been found particularly helpful to indicate that one method must still be active (i.e. on the call stack) while another method is being invoked (cf. section 5.7.1).

The JPDD shown in Figure 6.6 selects all invocations of methods whose names begin with "print" and which take an arbitrary number of arguments and which are addressed to an instance of class "OutputDecorator" (see message identified with "?jp"). The JPDD makes use of an indirect message and indirect activation bars in order to denote that these selected method invocations must occur in the control flow[48] of another method invocation of a method whose name begins with "print" and which takes an arbitrary number of arguments, too, and which must be addressed to the same instance of class "OutputDecorator" as the selected method invocation (see Figure 6.6).

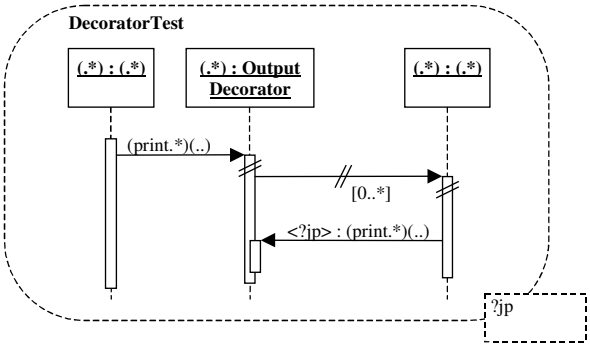


Figure 6.6 JPDD representing the join point selection of the decorator test aspect.

[48] at some deeper call stack level.

6.2.2.2 Comparing the JPDD with the Pointcut Implementation using Tracematches

When comparing the JPDD shown in Figure 6.6 to the Tracematch implementation discussed in section A.2 (Appendix A), one can observe that the Tracematch implementation of the join point selection defines two variables (i.e. "decorator" and "jp"; see line 21 in Listing 6.14) which are used to specify dependencies between the symbols of the Tracematch. The first variable "decorator" is used to require that all system events complying to the symbols must be issued on the same "OutputDecorator" instance (see `target` keyword in lines 24, 28, and 31; highlighted in Listing 6.14). The second variable "jp" is used to require that symbol "beginPrint" and symbol "endPrint" must refer to the same join point (made accessible via keyword `thisJoinPoint` in lines 25+32; highlighted by Listing 6.15). The latter dependency denotes that symbol "beginPrint" refers to the instant before the execution of the join point, while symbol "endPrint" refers to the instant after the execution of that join point. Finally, the Tracematch defines a regular expression which selects all execution traces where symbol "recurringPrint" occurs after "beginPrint" and where no symbol "endPrint" must occur in between those two symbols (see line 34 in Listing 6.16)[49].

The problem about the Tracematch implementation of the join point selection is that the free variable mechanism of Tracematches – which is designed to specify data constraints such as the one expressed by variable "decorator" (see Listing 6.14) – is used to specify a control flow dependency (using variable "jp"; see Listing 6.15)[50]. This may be problematic because readers of the join point selection may not expect this, and thus may easily misinterpret the control flow dependency as an ordinary data dependency.

Moreover, the control flow dependency involves a symbol which does not appear in the regular expression of the Tracematch. Readers of the join point selection need to be aware of the semantic implications of this "non-appearance" (i.e. that no system event must appear in the (tail of) the execution trace which complies to the selection constraints of that symbol; see Listing 6.16). In conclusion, readers of the Tracematch need to recognize first that there must exist a control flow dependency between two of the symbols of the Tracematch, just to find out later that this dependency must not take effect in the execution trace for the join point selection to match (see Listing 6.16).

Other than the Tracematch, the JPDD outlines the data dependency using a single lifeline symbol (i.e. "(.*) : OutputDecorator") and represents the control flow dependency with the help of indirect messages and indirect activation bars. That distinguished representation of the dependencies is expected to help readers of the join point selection to perceive that the recurrent invocations of the "print" method are supposed to be issued on the same "OutputDecorator" instance and furthermore that these recurrent invocations need to come to pass in nested order. It is expected that the different representation helps readers of the join point selection to recognize the different nature of these dependencies more easily.

---

[49] Recall that a symbol which is defined by the Tracematch yet which is not mentioned in the regular expression must *not* occur in the execution trace of a program for the Tracematch to match that execution trace.

[50] Refer to section A.2 (in Appendix A) to find out why no `cflow` pointcut designator may be used to realize the join point selection of the decorator test aspect.

Listing 6.14 Detecting the data dependency in the Tracematch.

```
3   Object tracematch(OutputDecorator decorator, JoinPoint jp) {
4
5     sym beginPrint before :
6       call(* *.print(..)) && target(decorator)
7       && let(jp, thisJoinPoint);
8
9     sym recurringPrint around(output) :
10      call(* *.print(..)) && target(decorator);
11
12    sym endPrint after :
13      call(* *.print(..)) && target(decorator)
14      && let(jp, thisJoinPoint);
15
16  beginPrint recurringPrint
      [...]
20  }
```

Listing 6.15 Detecting the control flow dependency in the Tracematch.

```
3   Object tracematch(OutputDecorator decorator, JoinPoint jp) {
4
5     sym beginPrint before :
6       call(* *.print(..)) && target(decorator)
7       && let(jp, thisJoinPoint);
8
9     sym recurringPrint around(output) :
10      call(* *.print(..)) && target(decorator);
11
12    sym endPrint after :
13      call(* *.print(..)) && target(decorator)
14      && let(jp, thisJoinPoint);
15
16  beginPrint recurringPrint
      [...]
20  }
```

Listing 6.16 Detecting the chronological dependencies in the Tracematch.

```
3   Object tracematch(OutputDecorator decorator, JoinPoint jp) {
4
5     sym beginPrint before :
6       call(* *.print(..)) && target(decorator)
7       && let(jp, thisJoinPoint);
8
9     sym recurringPrint around(output) :
10      call(* *.print(..)) && target(decorator);
11
12    sym endPrint after :
13      call(* *.print(..)) && target(decorator)
14      && let(jp, thisJoinPoint);
15
16  beginPrint x recurringPrint
      [...]
20  }
```

6.2.2.3 Summary

In summary, the JPDD shown in Figure 6.6 is considered to improve over the Tracematch implementation of the join point selection in that it explicitly distinguishes between the control flow constraint, which is highlighted by means of an indirect message and indirect activation bars, and the data constraint, which is expressed by means of a singular lifeline object. As a result, readers of the join point selection are expected to heed the particular relevance of the control flow to the overall goal of the join point selection more easily. Moreover, the distinguished representation of the constraints should prevent them from erroneously interpret the control flow dependency as a conventional data dependency. There is no need to carefully study the details of the pointcut implementation, nor to acquire a fundamental understanding of the reflective capabilities of the underlying programming language in order to detect such control flow dependencies. Likewise, readers of the join point selection are freed from inspecting the recurrent use of variable names in the `target` keywords of the symbols in order to discover that the system events designated by these symbols must be invoked on the same target instances.

The interaction diagram-based JPDD shown in Figure 6.6 is deemed to be particularly helpful to facilitate the comprehension of the join point selection considered here since it emphasizes the control flow-oriented conceptual view on program execution of the join point selection. That is, the JPDD highlights the nesting of method calls which is essential to the join point selection. It is expected that the explicit emphasis of this dependency in the interaction diagram-based representation of the join point selection (shown in Figure 6.6) helps readers to detect the key objectives of the join point selection, and thus makes comprehension of the join point selection easier – in particular in comparison to a representation where program execution is conceived as a sequence of atomic actions or events, and where nesting of system events must be realized by imposing auxiliary data dependencies between those actions or events (such as it is the case in the Tracematch implementation of the join point selection).

6.2.3 The Caching Aspect Implemented With Perl Aspect

The last example is about a caching aspect, which is implemented using Perl Aspect in section A.3 of Appendix A. The goal of the caching aspect is to prevent the repetitious execution of a complex computation and to re-use the computation results of an earlier execution, instead. The concrete caching aspect presented in Appendix A caches the repetitious computation of strongly connected components of a graph. Note that the implementation of the caching aspect makes use of two advice (rather than just one), both of which hook onto the same join point. By these means, the programmer of the aspect intends to emphasize that the aspect is about a repetitious invocation of the same method (rather than about an alternative processing of a singular invocation).

6.2.3.1 Using JPDDs to Represent the Join Point Selection

In order to represent the join point selection of the caching aspect using a JPDD, the underlying conceptual view on program execution needs to determined first. Considering that the goal of the join point selection is to intercept repeated invocations of a method with the same parameters, the join point selection can be accounted to refer to a data flow-

oriented view on program execution. Accordingly, the join point selection shall be represented using an activity diagram-based JPDD.

The corresponding JPDD is shown in Figure 6.7 and has already been explained in section 5.9.2 (Chapter 5). As join points, the JPDD selects all recurring invocations of a method with name "getScc" which are targeted at the same instance of class "graph" and which take the same object (of any type) as argument. Note how the JPDD makes use of an indirect control flow arrow in order to indicate that the recurring invocations of method "getScc" do not need to occur right after each other. The return value of the earlier invocation of method "getScc" is given an identifier "?scc" and is exposed by the JPDD by appending it to the export parameter box at the lower right corner of the JPDD.

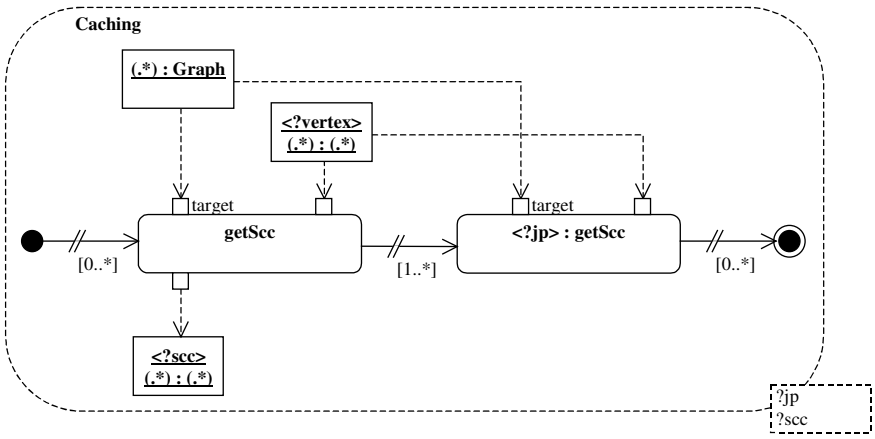


Figure 6.7 JPDD representing the join point selection of the caching aspect.

6.2.3.2 Comparing the JPDD with the Pointcut Implementation in Perl Aspect

When comparing the JPDD shown in Figure 6.7 to the Perl Aspect implementation discussed in section A.3 (Appendix A), one can observe that the Perl Aspect implementation of the join point selection implements the data dependency with help of a designated data structure (see Listing 6.17). That data structure is initialized by one advice specification (see lines 12-13) and it is evaluated by the other advice specification (see line 21). Fundamental knowledge about the used data structures (i.e. about hashes, in this case) is required in order to understand the initialization and evaluation of that data structure in the advice specifications.

Apart from that, readers of the Perl Aspect implementation need to be aware of the execution order of advice. As shown in Listing 6.18, both advice refer to the same pointcut (see lines 16+28). Furthermore, both advice invoke the intercepted original behavior (by calling `$context->run_original` in lines 10+25). Finally, both advice operate on the same data structure (as already mentioned above). In consequence, the execution order of the advice is significant because different execution orders lead to different behavior of the aspect. However, there is no programmatic construct in the Perl Aspect implementation which could indicate which of the advice executes first[51].

[51] Note that the execution order of advice in Perl Aspect is different from (i.e. converse to) the execution order of advice in AspectJ.

Listing 6.17 Defining and accessing data structures in Perl Aspect.

```
 5   my %chache = ();
 6
 7   before {
     [...]
12     $chache{$context->self} = {} unless $chache{$context->self};
13     $chache{$context->self}->{$context->params->[1]} = $return_value;
     [...]
16   } $pointcut;
17
18   before {
       [...]
21     if ($chache{$context->self} && $chache{$context->self}->{$context->params->[1]}) {
         [...]
27     }
28   } $pointcut;
```

Listing 6.18 Implicit execution order of advice in Perl Aspect.

```
 3   my $pointcut = call qr/Graph::.*strongly_connected_component_by_vertex$/;
     [...]
 7   before {
       [...]
10     my $return_value = $context->run_original;
       [...]
16   } $pointcut;
17
18   before {
       [...]
25       my $return_value = $context->run_original;
         [...]
28   } $pointcut;
```

In addition to that, some of the design decisions realized in the Perl Aspect module are somewhat "unconventional", e.g. the possibility to implement `around` behavior with help of `before` advice (see lines 10+25 in Listing 6.18), or the possibility to prevent the execution of the intercepted behavior by initializing a return value in the advice body (see line 23 in Listing 6.19). Such design decisions may perplex readers of the pointcut implementation because they are (at least partly) in contrast to the usual conventions which may be found in other aspect-oriented programming languages. Consequently, readers may be unsure and/or bewildered by the semantic implications of such design decisions.

Listing 6.19 Before advice implementing around behavior in Perl Aspect.

```
18   before {
       [...]
22       #join point adaptation:
23       $context->return_value($chache{$context->self}->{$context->params->[1]});
       [...]
28   } $pointcut;
```

In contrast to that, the JPDD shown in Figure 6.7 makes use of two distinct method symbols (with the same label, i.e. "getScc") and connects them using an indirect

relationship in order to highlight the repetitious invocation of the (same) method. Furthermore, it represents the involved data objects with the help of two object symbols (i.e. "(.*) : Graph" and "(.*) : (.*)"), which are connected to the method symbols in order to emphasize the repeated use of the (same) data objects in the repeated invocations of the method. Hence, other than the Perl Aspect implementation, the JPDD is very explicit about the chronological order in which the two method invocations need to occur and moreover about the data dependencies which must be fulfilled for the second invocation to denote a join point.

6.2.3.3 Summary

In summary, the JPDD shown in Figure 6.7 improves over the Perl Aspect implementation of the join point selection because it frees readers of the join point selection from the burden to contemplate about the runtime behavior of the program code. That is, readers of the join point selection do not need to reconstruct the chronological dependencies and data dependencies which arise from the inherent (and thus hidden) semantics of the programming language and/or from the various accesses to the shared data structure. As a result, the key objectives of the join point selection should be easier to grasp.

The use of an activity diagram-based JPDD is considered to be particularly helpful in that regard because it suits the data flow-oriented view on program execution which underlies the join point selection. That is, it emphasizes the role of the two objects that need to be involved in the repeated method invocations in order to make the join point selection match a join point. This is in contrast to a notation which requires a data structure to implement this requirement (such as Perl Aspect).

6.3 Conclusion: JPDDs As A Common Communication Means

6.3.1 Benefits

The previous subsections have illustrated that JPDDs are suitable to help developers with the comprehension of complex join point selections because they are capable of expressing the most relevant join point selection criteria in an explicit manner, such that the risk of misconceiving them is substantially reduced. This is in clear contrast to the textual counterparts of those JPDDs where readers are frequently forced to reconstruct the key selection constraints by carefully investigating the program code and by contemplating about its semantic implications (i.e. about the runtime effects which are going to arise during program execution). As illustrated by the examples in this chapter, this burden particularly arises if join point selections involve complex selection constraints which pertain to multiple system events, such as data constraints or chronological constraints.

Other than JPDDs, textual implementations of such join point selections commonly fail to externalize these data dependencies and/or chronological dependencies, e.g. with help of dedicated programming language constructs. In consequence, readers of the join point selections need to discover these dependencies "manually" from the semantic implications of the program code, e.g. such as they result from the shared usage of a common data structure. This obligation substantially impedes the detection and comprehension of those dependencies. Moreover, before readers are able to recognize and understand these dependencies at all, they are urged to acquire essential knowledge about

the data structures being used and their interface as well as about the semantic ideas, concepts, and mechanisms which underlie the programming language constructs (which are used in the pointcut implementation).

The previous subsections have illustrated how different these data structures, their interfaces, as well as the underlying ideas, concepts, and mechanisms of programming languages may be. Illustrative examples of the variation in the definition and usage of data structures are the way to define complex data structures in COBOL with help of "levels", the way to store and retrieve key-value-pairs to a map from the Standard Template Library (SLT) in C++, or the way to use (i.e. reference and dereference) hashes in Perl. Illustrative examples of the variation in the underlying ideas, concepts, and mechanisms in different programming languages are the absence of a local data storage in COBOL, the need to cope with context switches in C++, logic unification in PROLOG, or the "unexpected" way to define `around` behavior with help of `before` advice in Perl Aspect. All of these examples hamper the comprehension of the implementation of a complex join point selection in the respective programming language in so far that readers need to be aware of it and/or get familiar with it before they are able understand the implementation properly. By using JPDDs to represent the same join point selections instead, readers are freed from acquiring that necessary knowledge about a particular programming language – which means a significant relief to the software developer, in particular in situations in which they are not familiar with the programming language used to implement a join point selection and in which all they want is to get to know the key objectives of that join point selection.

Yet, even in situations where readers are well familiar with the programming language used to implement a join point selection, the burden of scrutinizing the program code in order not to overlook any implicit dependencies remains. This task may get very intricate because the (sub)components of a pointcut implementation may be arranged in arbitrary ways and variable names may be chosen randomly. Hence, readers of the pointcut implementation need to be very attentive in order to get a hold of all dependencies that exist in the program code. Note that programming idioms and naming conventions may alleviate parts of the problem for software developers working on the same software project – yet, they will not help the new software developer which is unaware of them.

Similar problems also remain if chronological dependencies as well as data dependencies are explicitly mentioned in the program code (as it is the case with aspects implemented in Alpha or Tracematches; see sections 3.1.4 and A.2). This is because readers of the program code need to mentally map the numerous occurrences of different variable names in the right manner in order to not to mix up such interdependencies. Hence, there is a risk that readers of the join point selection get confused about which variable is involved in which dependency. In contrast to that, JPDDs aggregate all occurrences of the same variable and represent that variable using a single symbol, which is subsequently related to all places where it is used, and which should therefore facilitate the proper identification of all dependencies in which the variable is involved.

In summary, JPDDs can be attested to help readers of complex join point selections to comprehend the key selection criteria of those complex join point selections for the following reasons:

- JPDDs externalize the dependencies that must exist between the system events which are relevant to a complex join point selection. While doing so, JPDDs represent singular matters with help of singular symbols.

- JPDDs are able to represent complex join point selections in a uniform and programming language-independent way, i.e. they use uniform symbols to represent similar selection constraints which may be implemented differently in various aspect-oriented programming languages.

As such, JPDDs are considered to significantly reduce the knowledge sharing problem postulated in the problem statement of this thesis (see Chapter 3).

6.3.2 Costs

It must be emphasized that the benefits of JPDDs come at a cost. This cost results from the necessity to learn the notational means of JPDDs, themselves. It is assumed, however, that these learning efforts are comparable to, if not less than, the learning efforts required to acquire a particular aspect-oriented programming language or its base language (e.g. its data structures and their interfaces, etc.). It is assumed that this is particularly true for software developers who are familiar with the UML, or with other related notations, such as state machines [*Hopcroft et al. (2007)*], state charts [*Harel (1987)*], Message Sequence Charts (MSC) [*ITU (1999)*], or the Business Process Modeling Notation (BPMN) [*OMG (2009)*], etc.

Furthermore, it is expected that these costs will pay off every time a software developer needs to cope with a new complex join point selection. This is because the software developers do not need to search manually for the key selection constraints in the details of the program code. They do not need to cope with shared data structures and their interfaces. And they do not need to deal with new programming language constructs which they are not familiar with. In these situations, JPDDs may serve as a common communication means which may be used by a community of software developers to communicate their complex join point selections to each other – without being forced to explain or learn the fundamentals of their respective programming language first, nor the patterns, idioms, or conventions which have been used to implement the join point selection, nor do developers need to point out or investigate the intricate interdependencies which result from the details of the program code. Hence, JPDDs should be a facilitator to knowledge sharing among software developers – both using the same programming language as well as across different programming languages.

6.3.3 Outlook to Next Chapter

While this chapter has elucidated the reasons why JPDDs are expected to facilitate the comprehension of complex join point selections, the next section is going to investigate this expectation with help of an empirical study.

Chapter 7
Evaluation

This chapter evaluates empirically if JPDDs facilitate the comprehension of complex join point selections. The goal is to show that the estimated benefits discussed in the previous chapter are actually achieved. To do so, the chapter first presents initial considerations about the overall motivation and the particular focus of the empirical evaluation (7.1). Then, it develops concrete research questions which are to be investigated by the evaluation (see section 7.2). Subsequently, a null hypothesis is formulated (based on the research questions) and an experiment design is described which is used to test the hypothesis (see section 7.3). Then, the details of the execution of that experiment are outlined (see section 7.4). Finally, the experiment results are analyzed and interpreted (see sections 7.5 and 7.6). The chapter concludes with a summary of the findings and an outline of the possible threats to the validity of the experiment (see sections 7.7 and 7.8).

7.1 Initial Considerations

It has been one of the objectives of the previous chapters to illustrate that many aspect-oriented programming languages require software developers to implement their join point selections manually and with the help of sophisticated workarounds. It has been exemplified why these manual and complex workarounds may impede the comprehension of a join point selection (see Chapter 3). In response to that, JPDDs have been presented as a means to represent join point selections in a concise and programming language-independent way which may help developers to understand what a join point selection is actually about (see Chapter 5).

Chapter 6 has exemplified the reasons why JPDDs are expected to improve over their textual counterparts with the help of several examples. While the explanations presented in that chapter may sound plausible and seem to justify the expectations, it remains hypothetical if the expected comprehension benefits will actually be attained by the readers of a join point selection. In order to verify if this is the case, an empirical investigation is in order in which participants actually need to comprehend join point selections, and which compares the efforts needed to comprehend a join point selection represented as a JPDD with the efforts needed to comprehend a join point selection specified using an aspect-oriented programming language.

The particular challenge of conducting such empirical investigation is to assure that the observed differences actually result from the different notations being used rather than from something else (e.g. from an unequal complexity of the join point selections under test) since only then the observed benefits can be attributed to the characteristics of the different notations. That is why this thesis conducts a **controlled experiment**, i.e. an empirical experiment which tries to fix as many influencing factors as possible (except the one referring to the different notations) in order to acquire as reliable evidence as possible

which confirms that the expected differences in the comprehension effort are actually achieved and that they are really due to the different notation.

Fixing all possible influencing factors in a controlled experiment (generally referred to as "confounding factors") is no trivial task, though, and strongly depends on the matter-of-interest that shall be investigated. Consequently, the matter-of-interest addressed by the controlled experiment needs to be well defined. That is why the controlled experiment in this thesis shall not be conducted for all of the previously presented notations of JPDDs, but just for one. Furthermore, it shall not cover all selection constraints that may be represented with that notation, but just one. Finally, the controlled experiment shall investigate the particular benefits that the *visual* notation of JPDDs bring forth to the comprehension of join point selection, as this is a distinguishing characteristic of JPDDs in comparison to the purely textual representation of the pointcut implementations.

In order to assure a fair comparison between the visual notation of JPDDs and its textual counterparts, an aspect-oriented programming language shall be selected that is capable of highlighting the relevant selection constraints of a join point selection in a comparably explicit and succinct manner as JPDDs do. This is because the benefits of an explicit representation of a selection constraint over an implicit definition of that selection constraint (hidden in the semantic details of program code) seems all too foreseeable. Apart from that, such comparison would not clarify if a potentially measured benefit results from the fact that the selection constraint is *explicitly* represented in the pointcut specification, of if it results from the fact that the selection constraint is *visually* represented in the pointcut specification.

In section 4.1 (Chapter 4), several aspect-oriented programming languages have been presented which provide likewise explicit means to represent *specific* join point selection constraints. The problem of these languages is that they usually only provide very specific join point selection means (i.e. they do not support all of the join point selection means offered by JPDDs). Apart from that, they are usually tied to a single base language which renders them useless to software developers of other base languages. Nevertheless, one of such languages shall be chosen to evaluate the visual notation of JPDDs. This is because a language providing an explicit representation of a particular kind of selection constraint is considered suitable enough for the purpose of evaluating the ability of JPDDs to ease the comprehension of that particular kind of selection constraint – even if the language may show insufficiencies to express other kinds of join point selection constraints.

It remains to determine which of the visual notations of JPDDs (i.e. the interaction diagram-based, the activity diagram-based, or the state chart-based notation) shall be evaluated. Furthermore, it needs to be determined what particular join point selection constraint shall be considered. Finally, an aspect-oriented programming language must be chosen which shall be compared to the JPDDs. For the controlled experiment presented in this thesis, these decisions are mostly driven by the capabilities of the existing aspect-oriented programming languages to explicitly represent selection constraints. Among these, the capabilities of Tracematches [*Allan et al. (2005)*] to explicitly represent data constraints in aspect-oriented join point selections have been found likewise explicit and succinct as the capabilities of JPDDs to express such data constraints using activity diagram-based JPDDs. And that is why the controlled experiment conducted in this thesis has been decided to compare activity diagram-based JPDDs and Tracematches with respect to their ability to facilitate the comprehension of data constraints in aspect-oriented join point selections.

Note that with the decision to evaluate just one selection constraint represented in just one notation using a controlled experiment, this thesis has decided to investigate if an improved comprehensibility of a selection constraint is really due to (i.e. caused by) the characteristics of the notation presented here. As such, this thesis has opted against a more comprehensive investigation covering all notational means of JPDDs, which might have been less thorough and thus would have left doubts if an observed difference in the comprehensibility of the selection constraints is really due to the characteristics of the notation. In other words, the goal of the empirical evaluation is decided to provide a strong support of the benefits of JPDDs in a very concrete situation rather than to provide weaker support of the benefits of JPDDs in general. The empirical evaluation conducted in this chapter is thus considered an adequate evaluation of JPDDs, even though further studies are needed to investigate the benefits of the other notations provided by JPDDs as well as their capabilities to represent other selection constraints.

7.2 Experiment Objectives

The objective of the experiment is to investigate if JPDDs – as opposed to Tracematches – facilitate the detection and comprehension of data constraints in complex join point selections. In order to do so, it is necessary to determine how this ability can be assessed. Therefore, it is the subject of this section to discuss possible intellectual efforts that are required to detect and comprehend data constraints in join point selections with the help of an example, and thus to define concrete research questions which are going to be addressed by the experiment.

The example is adopted from [*Allan et al. (2005)*] and is about ensuring the safe use of iterators. That is, it defines a Tracematch which intercepts any retrieval of an object from an iterator whose underlying data source has been modified. Note that the example has already been dealt with in the problem statement (see section 3.1.2) and in the discussion (see section 6.1.2) of this thesis. In comparison to [*Allan et al. (2005)*], the example has been slightly modified insofar that the Tracematch logs the object whose update has lead to the invalidation of the iterator, before a new ConcurrentModificationException is thrown. Figure 7.1 shows a corresponding JPDD of the join point selection and Listing 7.1 shows a corresponding Tracematch.

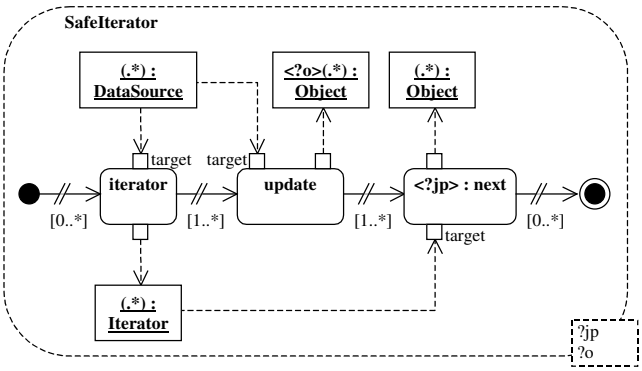


Figure 7.1 A sample JPDD.

The JPDD shown in Figure 7.1 selects all method invocations (`<?jp>`) of a method called "next" which are invoked on a target instance of type "Iterator". That target instance

of type "Iterator" must originate from a method call to a method named "iterator" which must be invoked on a target instance of type "DataSource". Furthermore, there must be an invocation of a method named "update" (addressed to the same instance as the invocation of method "iterator") which occurs in between the two method calls (of method "iterator" and of method "next"). Apart from the selected method invocation (<?jp>), the JPDD exposes the object (<?o>) which is returned by the invocation of method "update".

Listing 7.1 A sample Tracematch.

```
1 tracematch (Iterator i, DataSource ds, Object o) {
2   sym createIterator after returning(i):
3     call(Iterator DataSource.iterator())
4     && target(ds);
5   sym callNext before:
6     call (Object Iterator.next())
7     && target(i);
8   sym updateSource after returning(o):
9     call(Object DataSource.update(..))
10    && target(ds);
11
12  createIterator callNext* updateSource+ callNext
13  {
14    MyLogger.log(this, new Object[]{o}, true);
15    throw new ConcurrentModificationException();
16  }
17 }
```

The Tracematch shown in Listing 7.1 defines three symbols (`sym`), one for each of the relevant system events. The first symbol designates all invocations of method "iterator" which are declared by a class named "DataSource" and which return an object of type "Iterator". The second symbol selects all invocations of method "next" declared by a class named "Iterator" which return a value of type "Object". Finally, the third symbol designates all method calls of method "update" which are invoked on instances of type "DataSource" and which return an instance of type "Object". Apart from the symbols, the Tracematch defines three variables in its signature (i.e. `i`, `ds`, and `o`) which are used to expose objects from the symbols as well as to establish data dependencies between the symbols. For example, the return value of the first symbol must be identical to the target instance of the second symbol; furthermore, the target instance of the first symbol must be identical to the target instance of the last symbol. Finally, the Tracematch defines a regular expression which determines the sequential order in which the symbols must occur in the execution trace of a base program.

Note that the JPDD and the Tracematch shown in Figure 7.1 and in Listing 7.1 make use of different styles to express data constraints: the JPDD represents data dependencies using directed edges and the Tracematch makes use of special keywords (such as `target` or `returning`) in order to indicate how objects are involved in which methods.

In order to detect and understand these data constraints, readers of both JPDD and Tracematch need to discover that the target objects of the method calls to method "iterator" and method "update" need to be the same. Likewise, they need to discover that the return value of the method call to method "iterator" must be identical to the target object of the method call to method "next". Finally, readers need to discover that the

return value of the method call to method "update" is merely exposed to the advice and is not involved in any data dependency.

In more general terms, readers of the join point selection need to (a) identify all dependent objects, i.e. all objects which are involved in more than one method. Furthermore, they need to (b) recognize in how many methods and (c) in which methods the objects are involved, as well as (d) how they are involved (e.g. as input or output parameter). Another issue is (e) the identification of those methods which involve same objects.

In summary, the following research questions can be formulated which shall be addressed by the experiment:

Do ***JPDD****s – as opposed to* ***Tracematches*** *– help readers of complex join point selections*

Q1. *with the identification of dependent objects, i.e. of objects which are involved in at least two methods?* (This question relates to (a) and (b).)

Q2. *with the detection of objects which are involved in more than two methods?* (This question relates to (b).)

Q3. *with the identification of the methods in which a particular dependent object is involved?* (This question relates to (c).)

Q4. *with the identification of methods which involve a particular common dependent object?* (This question relates to (c) and (e).)

Q5. *with the identification of methods which involve several common dependent objects?* (This question relates to (e).)

Note that the experiment does not take into account *how* the objects may be involved in the methods (e.g. as input or output parameters). (This question would relate to (d).)

Having identified the research questions, it is necessary to contemplate possible factors which may have an effect on the efforts needed to solve these questions. Apart from the notation used to represent the join point selections (which is the factor whose effects shall be investigated here), another possible influencing factor could be the complexity of the join point selections. This is because it is reasonable to assume that finding the answers to research questions Q1-Q5 will require more efforts when considering a "complex" join point selection than when considering an "easy" join point selection. Another influencing factor could be the familiarity of the reader with the notation used to represent it. This is because it is plausible to assume that an experienced reader will find his/her way through the pointcut specification faster than an inexperienced reader. In a controlled experiment, such influencing factors (often referred to as "confounding factors") are tried to be reduced, or counterbalanced, as much as possible in order to prevent them from influencing (or "confounding") the experiment results, and thus in order to be able to attribute any difference observed in the experiment results to the different treatments of the factor-under-study (i.e. the notation, in this case).

It is subject of the next section to outline the design of the controlled experiment conducted in this thesis. In particular, it is part of the section to elucidate what measures are taken in order to control the confounding factors such that their effects on the experiment results are counterbalanced as much as possible.

7.3 Experiment Design

The first step of an experiment design is to define two hypotheses: a "null hypothesis" (H0) stating that the factor-under-study has *no* effect, and an "alternative hypothesis" (H1) stating that the factor-under-study *has* an effect. The goal of the experiment is then to show that the null hypothesis is wrong, and to conclude from this observation that the alternative hypothesis must be true.

Hence, based on the research questions Q1-Q5 formulated in section 7.2, the hypotheses of this experiment can be formulated as follows:

H0. *There is* **no** *difference between* ***JPDDs*** *and* ***Tracematches*** *with respect to the comprehensibility of data constraints in join point selections.*

H1. *There* **is** *a difference between* ***JPDDs*** *and* ***Tracematches*** *with respect to the comprehensibility of data constraints in join point selections.*

To test these hypotheses, several join point selections (from now on referred to as "test examples") are presented to several participants (divided into two groups) who are asked to correctly respond to a single question about each test example. There are two kinds of questions (later referred to as "tasks") with several variants, yet only one kind of question is asked about each test example.

The test **measures** the time to correctly answer a question. The timer starts when a test example is shown to the participants. The timer stops when the participant has found the right answer to the question. Participants are allowed to submit wrong answers. Whenever they do so, they are told so and they are given another chance to submit the right answer. No penalty is added to the measured time for wrong answers.

The study realizes a **within-subject design**, which means that all participants need to answer the same question about the same test example once for each notation. The motivation of that is to counterbalance any confounding impact that might arise from the capabilities of the participants, such as background knowledge, reading speed, typing speed, etc.. In order to prevent participants from remembering a test example when they see it a second time in the other notation, different method and object names are used in either notation. The names are generated automatically, and they follow a "consonant-vowel-consonant-vowel" pattern in order to make them equally "comprehensible". Apart from that, test examples remain unchanged. Insofar, the test examples are assumed to be "equivalent" (from the perspective of comprehensibility) in both notations.

The questions Q1-Q5 mentioned in section 7.2 have been translated into two kinds of concrete **tasks** (each having several variants). In contrast to the questions mentioned in section 7.2, these tasks are intentionally designed to be very crisp about what subjects need to do in order to solve them. The motivation is to reduce the time needed to read and understand a question and to deduce how it can be solved (as this could confound the measured response times). Accordingly, participants are asked to perform either of the following tasks/variants on a given test example:

T1. "*Please click (in arbitrary order) on all objects which are involved in at least* n *methods*", where n equals 2, 3, 4, or 5.

T2. "*Please click (in arbitrary order) on all methods which objects* o_1*, ..., and* o_n *are involved in*", where o_1, ..., o_n are names of objects shown in the test example, and where $|\{o_1, ..., o_n\}|$ equals 2, 3, or 4.

Task T1 relates to questions Q1 and Q2 (with different variants relating to Q1 (n=2) and Q2 (n=3, 4, or 5)). Task T2 relates to questions Q3, Q4 and Q5 (with different variants relating to Q3 and Q4 (n=1)[52] and Q5 (n=2, 3, or 4)).

Test **participants** are not expected to be familiar with either of the notation beforehand. Accordingly, they are instructed how to read the different notations prior to the test. The goal of the introduction is to enable participants to answer the test questions correctly. The goal is not to enable them to effectively specify join point selections with either of the notations. A pretest is used to assess if the participants are actually able to understand and solve the questions properly.

No hints or guidelines are given to the participants how to solve the task (guessing is permitted). No handouts or **additional material** are provided to the participants for solving the task, except for the overview shown in Table 7.1 (which was requested by one of the participants):

Table 7.1 Handout given to participants during the test.

	Tracematch	JPDD
objects	args(...) returning(...)	
methods	sym ...	

Test examples are intentionally designed to be rather "complex" (in particular, in comparison to the example shown in section 7.2). The motivation of this is that the test aims to investigate if there is a difference between the two notations with respect to the comprehensibility of data constraints, and it is assumed that comprehension gets more difficult when a join point selection grows more "complex". Therefore, it is believed that a possible difference is likely to become manifest when a join point selection is "complex" (in case it exists at all).

In concrete terms, this means that all test examples are equal with respect to the number of methods, objects, and method-to-object-relationships involved in them. That is, all test examples involve ten methods, ten objects, and 24 method-to-object-relationships. The 24 method-to-object-relationships are arbitrarily and non-uniformly distributed over methods and objects such that

P1. each object is connected to at least one method,

P2. each method is connected to at least one object (which may be an input parameter or an output parameter to the method),

P3. seven methods have (at least) an output parameter,

P4. each method is connected to at least one dependent object (which may be an input parameter or an output parameter to the method).

Adhering to these constraints, all test examples are considered equally "complex". Nevertheless, due to the non-uniform distribution of method-to-object-relationships, test examples may vary with respect to the number of dependent objects as well as with respect

[52] Note that Q3 and Q4 basically denote the same task.

to the distribution of their method-to-object-relationships (Table 7.3 and Table 7.4 give concrete values about the differences between the test examples in this experiment).

Test examples are presented to the participants in alternating notations, e.g. the first test example is shown using Tracematches, while the second test example is represented as a JPDD, and so forth (see Table 7.2 for an example). The motivation of this is to control learning effects which may result from the *order in which the notations are used* (thus, the goal of doing this is to increase the internal **validity** of the experiment). There are two groups of participants: one group of participants starts out with JPDDs, while the other group of participants starts out with Tracematches. Note that this does not mean that participants always see test examples in the same notation first (see test example "4755" in Table 7.2 for example).

The order of the test examples is the same for all participants. It only differs with respect to the notation which is used to visualize the test examples. Hence, participants of different groups do not see the same test example *in the same notation* at the same time during the test. Instead, they see the same text example *in different notations* at the same time

Table 7.2 Arbitrarily computed order of test examples in bunches of four.

T1	JPDDfirst (N=16)		TMfirst (N=15)	
1	4751	JPDD	4751	TM
2	4755	TM	4755	JPDD
3	4773	JPDD	4773	TM
4	4751	TM	4751	JPDD
5	5930	JPDD	5930	TM
6	5930	TM	5930	JPDD
7	4755	JPDD	4755	TM
8	4773	TM	4773	JPDD
9	4705	JPDD	4705	TM
10	5756	TM	5756	JPDD
11	4761	JPDD	4761	TM
12	4705	TM	4705	JPDD
13	4731	JPDD	4731	TM
14	4761	TM	4761	JPDD
15	5756	JPDD	5756	TM
16	4731	TM	4731	JPDD
17	6022	JPDD	6022	TM
18	5936	TM	5936	JPDD
19	5772	JPDD	5772	TM
20	6022	TM	6022	JPDD
21	5936	JPDD	5936	TM
22	5772	TM	5772	JPDD
23	4767	JPDD	4767	TM
24	4767	TM	4767	JPDD
25	4715	JPDD	4715	TM
26	4709	TM	4709	JPDD
27	4709	JPDD	4709	TM
28	4741	TM	4741	JPDD
29	5988	JPDD	5988	TM
30	5988	TM	5988	JPDD
31	4741	JPDD	4741	TM
32	4715	TM	4715	JPDD

T2	JPDDfirst (N=17)		TMfirst (N=18)	
1	2216	JPDD	2216	TM
2	2816	TM	2816	JPDD
3	2876	JPDD	2876	TM
4	2876	TM	2876	JPDD
5	discard	JPDD	discard	TM
6	discard	TM	discard	JPDD
7	2816	JPDD	2816	TM
8	2216	TM	2216	JPDD
9	2214	JPDD	2214	TM
10	discard	TM	discard	JPDD
11	2770	JPDD	2770	TM
12	2214	TM	2214	JPDD
13	2936	JPDD	2936	TM
14	2936	TM	2936	JPDD
15	discard	JPDD	discard	TM
16	2770	TM	2770	JPDD
17	2902	JPDD	2902	TM
18	discard	TM	discard	JPDD
19	discard	JPDD	discard	TM
20	2712	TM	2712	JPDD
21	2712	JPDD	2712	TM
22	2902	TM	2902	JPDD
23	discard	JPDD	discard	TM
24	discard	TM	discard	JPDD
25	2820	JPDD	2820	TM
26	2988	TM	2988	JPDD
27	2988	JPDD	2988	TM
28	2182	TM	2182	JPDD
29	2182	JPDD	2182	TM
30	2970	TM	2970	JPDD
31	2970	JPDD	2970	TM
32	2820	TM	2820	JPDD

during the test (see Table 7.2). The order has been randomly chosen. Care has been taken to ensure that paired test examples do not follow each other immediately all too often (since this may make participants suspect that they work on paired test examples despite of the different object and method names). Furthermore, paired test examples are grouped into bunches of four, and they are arranged such that all paired test examples of one bunch are presented to the participants before the first paired test example of the next bunch is shown (see Table 7.2). That way, learning effects arising from the *order of the test examples* should be counterbalanced after every eight test examples. Participants are not made aware of this bunching.

The **apparatus** used to measure the response time is a JavaScript application running in a Firefox internet browser. The only input device needed to select and submit answers is a pointing device. Participants may choose the device at their convenience (e.g. a mouse, touch pad, or pointing stick, etc.). However, they are requested not to switch the input device during the experiment. The measured times are submitted to a web application (running on a Tomcat web server) which stores the measured times in a MySQL database.

In summary, the experiment implements a three-factorial design with two within-subject variables ("notation" and "test example") and one between-subject variable ("alternation"). The treatments of the variable "notation" are "JPDD" and "TM"; the treatments of the variable "test example" are the ids of the text examples (see Table 7.2); and the treatments of the variable "alternation" are "JPDDfirst" and "TMfirst". Accordingly, the plan is to use a three-way ANOVA with repeated measurements on two variables as **analysis procedure**. The risk is that the measured data does not satisfy the assumptions of the ANOVA. Fortunately, the ANOVA has proven to be quite robust against violations of (at least some of) its assumptions – provided that the number of test participants in each of the groups is equal in size and reasonably large (cf. [*Gene V. Glass & Sanders (1972)*]). Violations of the other assumptions may be compensated, e.g. by the corrections suggested by Box [*Box (1954)*] or Greenhouse and Geisser [*Geisser & Greenhouse (1958)*].

7.4 Experiment Execution

For the test, a total of 32 of paired **test examples** has been automatically generated, i.e. 16 test examples for task T1 and 16 test examples for task T2, which makes it roughly 5±1 test examples for each variant of the tasks. A lower number of paired test examples has been generated for task T1/variant n=5 and for task T2/variant n=4 due to an abortion of the automatic generation procedure after approx. 40-50 unsuccessful attempts to generate further suitable test examples. Table 7.3 and Table 7.4 indicate how many test examples have been generated for task T1 and T2, respectively, and for each of their variants.

Unfortunately, four of the 16 automatically generated test examples of task T2 had to be discarded after the experiment (i.e. two questions of variant n=3 and two questions of variant n=4) because they turned out not to meet the initially posed question Q5 (because the set of objects $\{o_1, ..., o_n\}$ unintentionally involved an object which was used by a single method only). These test examples are marked in Table 7.2 as "discard(ed)".

Hence, participants answered a total of 56 questions (i.e. 16 questions about Tracematch examples relating to task T1, and 16 questions about their JPDD counterparts,

as well as 12 questions about Tracematch examples relating to task T2, and 12 questions about their JPDD counterparts). Questions have been posed in German.

Test examples have been automatically generated from bipartite directed graphs (consisting of object nodes and method nodes as well as directed edges between them), which have been generated automatically and randomly by the computer (this includes the generation of node labels, i.e. of object names and method names, following a "consonant-vowel-consonant-vowel" pattern). The assignment of test examples to tasks has been accomplished automatically, too. The layout of visual test examples has been automatically generated using the GraphViz tool [*Gansner & North (2000)*] and its dot algorithm. Figure 7.2 and Listing 7.2 give an example of a generated test example, once represented as a JPDD and once represented as a Tracematch, respectively (a bigger and more readable version of the JPDD is given in Appendix C).

Listing 7.2 A test example (of the pretest) presented as Tracematch.

```
tracematch (Object peno, Object tido, Object dawe, Object mafu, Object
sifu, Object nefi, Object nude, Object pado, Object pogo, Object delo) {
  sym dika after returning(pado):
    call(* *.dika(..));
  sym mole after returning(tido):
    call(* *.mole(..)) && args(pogo);
  sym mire after returning(sifu):
    call(* *.mire(..)) && args(pado, delo);
  sym wele after returning(mafu):
    call(* *.wele(..)) && args(dawe, delo);
  sym gase after returning(nude):
    call(* *.gase(..)) && args(dawe);
  sym wapu after returning(peno):
    call(* *.wapu(..)) && args(dawe);
  sym losi after returning(nefi):
    call(* *.losi(..)) && args(nude);
  sym none after:
    call(* *.none(..)) && args(nude, nefi, peno, pogo);
  sym liko after:
    call(* *.liko(..)) && args(pado, peno);
  sym sobo after:
    call(* *.sobo(..)) && args(delo, nefi, pogo);
  dika mole mire wele gase wapu losi none liko sobo
  {
  executeadvice(peno, tido, dawe, mafu, sifu, nefi, nude, pado, pogo,
delo);
  }
}
```

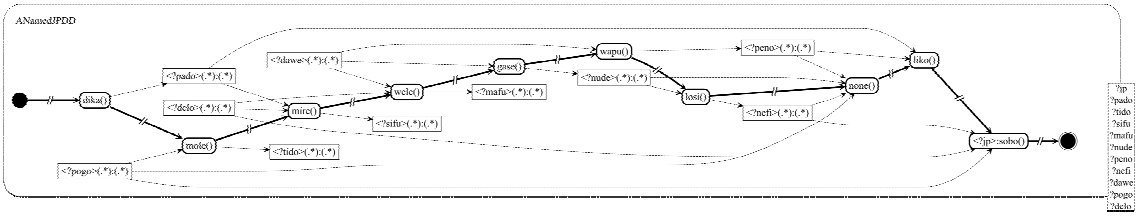


Figure 7.2 A test example (of the pretest) presented as JPDD.

Table 7.3 Variations in automatically generated test examples for task T1.

T1	characteristics of task	▪ variant (n)	▪ answer nodes	characteristics of test example	▪ dependent objects	▪ dependent output	▪ distribution of param.	characteristics of presentation order	▪ bunch	sequence no.	▪ 1st notation	▪ 2nd notation	▪ difference
4705		2	7		7	4	0.308		B2		9	12	3
4755		2	4		4	2	0.267		B1		7	2	-5
4767		2	6		6	4	0.200		B3		23	24	1
6022		2	7		7	4	0.258		B3		17	20	3
4709		3	6		6	4	0.292		B4		27	26	-1
4715		3	2		2	1	0.167		B4		25	32	7
4731		3	5		5	2	0.258		B2		13	16	3
4751		3	3		3	1	0.233		B1		1	4	3
4773		3	6		6	3	0.258		B1		3	8	5
5936		3	5		5	3	0.275		B3		21	18	-3
4741		4	2		2	0	0.208		B4		31	28	-3
5756		4	4		4	1	0.208		B2		15	10	-5
5772		4	3		3	2	0.225		B3		19	22	3
5988		4	4		4	2	0.225		B4		29	30	1
4761		5	2		2	0	0.142		B2		11	14	3
5930		5	2		2	0	0.175		B1		5	6	1

Table 7.4 Variations in automatically generated test examples for task T2.

T2	characteristics of task	▪ variant (n)	▪ answer nodes	characteristics of test example	▪ dependent objects	▪ dependent output	▪ distribution of param.	characteristics of presentation order	▪ bunch	sequence no.	▪ 1st notation	▪ 2nd notation	▪ difference
2214		1	5		4	1	0.208		B2		9	12	3
2216		1	3		9	6	0.233		B1		1	8	7
2816		1	5		3	3	0.225		B1		7	2	-5
2970		1	4		6	4	0.375		B4		31	30	-1
2988		1	4		6	5	0.292		B4		27	26	-1
2182		2	4		3	1	0.175		B4		29	28	-1
2820		2	3		5	3	0.167		B4		25	32	7
2876		2	2		7	5	0.208		B1		3	4	1
2936		2	2		5	3	0.192		B2		13	14	1
2770		3	2		8	6	0.383		B2		11	16	5
2902		3	3		3	2	0.308		B3		17	22	5
2712		4	1		6	5	0.208		B3		21	20	-1

All test examples satisfy the constraints P1-P4 defined in section 7.3. Hence, they are considered "equally complex". Nevertheless, due to the automatic generation process, they vary with respect to the following characteristics (see Table 7.3 and Table 7.4):

C1. the variant of the task which participants have to fulfill with the given test example (see column "variant" in Table 7.3 and Table 7.4),

C2. the number of nodes that need to be selected in order to solve the given task (see column "answer nodes" in Table 7.3 and Table 7.4),

C3. the number of objects which are involved in more than one method (see column "dependent objects" in Table 7.3 and Table 7.4),

C4. the number of objects which are involved in more than one method and which represent an output parameter to one of these methods (see column "dependent output obj." in Table 7.3 and Table 7.4),

C5. the distribution of the parameters over methods (indicated by a Gini coefficient), which indicates whether all methods have the same number of parameters (=0), or if only one method involves (all of the) parameters (=1)[53],

C6. the bunch which the test example belongs to (see column "bunch" in Table 7.3 and Table 7.4),

C7. the sequential position in which the test example is shown to the participants using the first notation, as specified by Table 7.2 (see column "1st notation" in Table 7.3 and Table 7.4),

C8. the sequential position in which the test example is shown to the participants using the second notation, as specified by Table 7.2 (see column "2nd notation" in Table 7.3 and Table 7.4),

C9. the difference between the aforementioned positions, which denotes the number of test examples after which participants work on the same test example again using the other notation (see column "difference" in Table 7.3 and Table 7.4; a negative value indicates that the participants work on the second notation first).

Note that in case of task T1, the nodes that need to be selected in order to solve the given task (i.e. the "answer nodes") always equates to the objects which are involved in more than one method (i.e. the "dependent objects"). This is a result of the automatic generation process of the test examples. Participants are not made aware of this fact. Note further that in both tasks T1 and T2, test examples are mostly – yet not always – first presented to the participants in the notation which the participants have started with, i.e. in their "first notation" (see positive values in column "difference" in Table 7.3 and Table 7.4).

A total of 35 **participants** has participated in the experiment. The participants have been recruited from undergraduate lectures in computer science. All of them were students (see Table 7.5 and Table 7.6). They have been invited to participate in the study voluntarily by announcements in the lectures and via mailings lists. Unfortunately, the announcements met with no response, and thus most of the participants have been personally addressed and invited again until they finally participated in the study.

[53] Note that due to the constraints P1-P4 defined in section 7.3, the Gini coefficient cannot grow greater than 0.575.

Participants have been divided into groups of (roughly) the same size for each task. The assignment of participants to groups has been accomplished automatically by the apparatus. The resulting structure of the groups is shown in Table 7.5 and Table 7.6: a total of 31 participants performed task T1, and a total of 35 participants performed task T2. All participants performing task T1 also performed task T2. Task T2 has been performed before task T1. The tasks have been accomplished on different days.

Table 7.5 Structure of the group performing task T1.

Task T1 (N=31)	JPDDfirst (N=16)	TMfirst (N=15)
degree=computer-science	16	15
skill=knowledged	2	2
skill=na	3	0
skill=novice	11	13
status=bs-student	16	13
status=ms-student	0	2

Table 7.6 Structure of the group performing task T2.

Task T2 (N=35)	JPDDfirst (N=17)	TMfirst (N=18)
degree=computer-science	17	18
skill=knowledged	2	3
skill=na	3	1
skill=novice	12	14
status=bs-student	17	16
status=ms-student	0	2

7.5 Experiment Analysis

In this section, the measured data is analyzed. The section starts out with a short summary of the basics of statistical hypothesis testing. Then, descriptive data about the experiment results is presented. Subsequently, the three-way ANOVA with repeated measurements is conducted. Finally, the data is explored in order to identify possible reasons for the test results.

Note that this section does not contain a thorough introduction to the statistical approaches being used. The interested reader may find such introductions in any standard textbook on inferential statistics (for this thesis, [*Bortz (1999)*] has been used). All data has been processed with R, SPSS, and Microsoft Excel.

7.5.1 Statistical Background

The goal of the statistical analysis of an experiment is to investigate if the empirically measured results contradict the formulated null hypothesis. This means in the case considered here, the goal of the statistical analysis is to investigate if a difference in response time between the JPDD representation of a join point selection and its textual counterpart cannot be observed.

To do so, the statistical analysis calculates a so-called **p-value** which denotes the conditional probability with which one would expect to measure the observed (or even more extreme) results, while assuming that the null hypothesis (H0) is true:

$$\text{p-value} = p(\text{observed results} \mid H0)$$

By convention, a null hypothesis is generally considered to be *not* true (i.e. it is *rejected*) if the p-value of the measured data is smaller than a certain threshold, called the "significance level" or **α-level**. The α-level refers to the "probability of error" with which a null hypothesis is rejected although it is true (such an – erroneous – decision is generally referred to as **α-error** or **Type I error**[54]). The α-level needs to be determined prior to an empirical experiment. For this experiment, p-values smaller than $\alpha=0.01$ are considered "significant", whereas p-values smaller than $\alpha=0.10$ are considered "close-to-significant".

It is important to note that p-value and α-level in fact refer to two different things, and that their comparison during a statistical analysis is based on mere convention. According to that convention, it is agreed to reject a null hypothesis if the conditional probability to obtain results like the ones that have been observed (under the condition that the null hypothesis is true) drops below the probability of error (to reject the null hypothesis although it is true) that one is willing to accept. Likewise, it is important to realize that the p-value does *not* determine the probability that the null hypothesis is true, i.e. p(H0), or that the alternative hypothesis is true, i.e. 1 – p(H0), or that the null hypothesis is true under the condition of the observed result, i.e. p(H0 | observed results), or that one may obtain results like the ones that have been observed (in the unconditional case), i.e. p(observed results).

In addition to p-values, the statistical analysis also calculates so-called **confidence intervals** which are used as interval estimates[55] for (unknown) population parameters (examples of population parameters are median, mean, standard deviation, etc.). Confidence intervals denote ranges of values which contain a certain percentage (e.g. 95% or 99%) of all possible "true" population parameters which could have "generated" the observed experiment results. In the statistical analysis conducted here, 99%-confidence intervals are calculated to estimate the "true" value of the means of response times. Likewise to p-values, confidence intervals may indicate the "significance" of the experiment results. In addition to that, though, confidence intervals may also reveal the overall tendency of the experiment results.

In order to properly calculate p-values and confidence intervals, many statistical approaches make assumptions about the measured data[56]. This is necessary in order to properly estimate the parameters of a population. The ANOVA which is going to be performed in this section, for example, assumes normality, homoscedasticy, and – since this is an ANOVA with repeated measurements on at least one variable – sphericity. **Normality** means that the measured data (i.e. the response times) must be normally distributed (for each test example). **Homoscedasticy** requires that the variances of the measured data must be equal for each group (i.e. for group "TMfirst" and for group "JPDDfirst"). **Sphericity** requires that the variances of the differences between all pairs of

[54] The α-error or Type I error can be contrasted to the β-error or Type II error which refers to the situation where the alternative hypothesis (H1) is rejected although it is true.

[55] Interval estimates can be contrasted to point estimates, which denote single values (such as median or mean).

[56] In fact, such assumptions apply to the overall population being examined, which – ideally – implies that the sample drawn from that population should possess the same characteristics.

treatments of the within-subject variables as well as of their "interactions" (i.e. between the treatments of variable notation, and between the treatments of variable test example, and between all combinations thereof) must be equal, too. In particular, the latter two assumptions are essential so that the marginal means of each (combination) of the treatments may be properly estimated.

Any violation of an assumption usually leads to an increase of the α-error or Type I error[57] and/or to a reduction of the power of the statistical approach. Fortunately, some violations can be corrected (to a certain extent) in order to sustain the quality characteristics of the approach (to a sufficient degree). An example of such correction which is used to correct a violation of the sphericity assumption in case of an ANOVA with repeated measurements is the correction suggested by Box [*Box (1954)*] or by Greenhouse and Geisser [*Geisser & Greenhouse (1958)*] (as it turns out, these corrections will be used later in this section). Other violations have shown to lead to negligible impacts only (provided that certain other prerequisites hold; cf. [*Gene V. Glass & Sanders (1972)*]).

After this very rough introduction to some of the very basics of statistical hypothesis testing, the statistical analysis of the experiment results starts out with a mere description of the experiment results. The actual hypothesis testing is performed afterwards.

7.5.2 Descriptive Data

Table 7.7 and Table 7.8 give important descriptive data, i.e. median and standard deviation (sd), about the measured response times, divided by task ({"T1", "T2"}), alternation ({"TMfirst", "JPDDfirst"}), and notation ({"TM", "JPDD"}). Table 7.9 and Table 7.11 show further descriptive data, i.e. mean and standard deviation (sd), in a transposed fashion. The tables underline minimum and maximum median and mean for each group and for each notation; overall minimum and maximum medians and means are additionally printed in bold. Figures visualizing the measured response times can be found in the Appendix C. All data refers to response time and is given in milliseconds (ms).

Looking at the descriptive data about task T1 shown in Table 7.7, it can be noticed that both median and standard deviation of the response times of task T1 are almost always much higher for Tracematches than for JPDDs. Moreover, for task T1, the standard deviation of the response times for JPDDs is notably low even in absolute terms, and it is pretty stable across test examples. At the same time, the median varies between test examples from roughly 6 seconds (for test example 5930, see group "TMfirst") to about 15 seconds (for test example 4773, see group "TMfirst"), which is 2.5 times more. For Tracematches, both median and standard deviation vary much more, with the median varying from about 9 seconds (for test example 5930, see group "JPDDfirst") to almost 133 seconds (for test example 4773, see group "TMfirst"), which is roughly 25 times more.

For task T2 (see Table 7.8), the variation of medians and standard deviations is more modest (than for the Tracematches in task T1). Furthermore, it seems that – this time – response times of Tracematches are more steady than response times of JPDDs (both in terms of median and standard deviation). For Tracematches, the median varies from 15 seconds (for test example 2970, see group "JPDDfirst") to almost 34 seconds (for test example 2902, see group "TMfirst"), which is approx. 2.3 times more. Whereas for JPDDs

[57] and/or to an increase of the β-error or Type II error.

Table 7.7 Descriptive data for task T1.

T1	TM		JPDD		Difference	
	median	sd	median	sd	median	sd
JPDDfirst (N=16)						
4705	57,782.5	39,737.15	13,157.5	4,508.36	**-41,892.0**	39,614.41
4709	55,754.5	35,843.41	14,271.0	23,312.49	**-38,624.0**	45,998.30
4715	10,829.0	3,225.76	8,633.5	4,186.79	**-4,110.0**	5,570.43
4731	26,446.5	31,354.24	10,573.5	2,925.23	**-13,888.0**	29,632.15
4741	12,206.5	6,598.82	7,744.0	2,186.01	**-2,443.0**	7,111.88
4751	11,440.5	5,280.52	8,824.0	3,416.25	**-3,047.0**	4,847.64
4755	20,858.5	15,687.96	11,387.5	2,242.95	**-9,110.0**	14,496.67
4761	12,706.5	6,125.22	5,948.5	1,416.58	**-6,353.0**	5,643.03
4767	38,861.5	32,510.55	12,094.0	7,668.81	**-24,789.0**	28,952.64
4773	97,423.5	60,380.24	14,168.5	5,224.36	**-75,085.0**	59,353.85
5756	35,914.5	27,807.28	11,263.5	5,520.80	**-22,162.0**	27,505.96
5772	20,643.0	8,757.46	7,044.5	3,077.92	**-12,899.0**	9,147.78
5930	**9,279.5**	7,467.50	7,075.5	1,646.94	**-2,310.0**	7,186.46
5936	57,524.5	66,499.38	9,720.5	13,163.67	**-43,362.0**	65,483.35
5988	21,141.5	14,939.35	9,971.5	6,916.65	**-10,326.0**	16,952.57
6022	26,376.0	28,853.29	11,052.0	2,112.48	**-14,415.0**	27,293.90
TMfirst (N=15)						
4705	64,003.0	37,719.37	11,957.0	8,670.19	**-40,799.0**	36,983.65
4709	82,006.0	64,053.72	13,200.0	4,613.79	**-61,996.0**	62,682.57
4715	16,366.0	14,437.40	8,308.0	2,761.87	**-7,866.0**	12,719.20
4731	46,957.0	20,908.07	10,210.0	2,693.26	**-37,016.0**	19,304.02
4741	16,246.0	13,269.16	6,660.0	2,882.13	**-10,938.0**	11,085.84
4751	25,164.0	13,413.31	8,204.0	2,785.56	**-16,754.0**	11,699.42
4755	35,748.0	13,824.20	11,627.0	2,721.58	**-21,745.0**	13,300.61
4761	11,929.0	13,651.77	7,403.0	2,853.24	**-5,541.0**	12,280.41
4767	54,864.0	65,250.31	10,595.0	4,918.96	**-44,381.0**	63,102.55
4773	**132,722.0**	92,296.74	**15,374.0**	5,421.25	**-117,348.0**	87,376.18
5756	49,859.0	33,056.60	11,264.0	7,698.75	**-32,060.0**	31,961.19
5772	35,890.0	17,482.82	7,384.0	3,496.89	**-28,919.0**	15,537.91
5930	20,797.0	19,715.32	**5,875.0**	2,398.35	**-13,745.0**	17,691.12
5936	48,780.0	38,563.50	9,072.0	4,076.73	**-41,380.0**	35,137.87
5988	42,734.0	21,295.62	10,998.0	5,564.37	**-26,996.0**	20,387.38
6022	53,790.0	29,281.13	11,212.0	3,471.91	**-37,789.0**	27,839.32

the median varies from almost 10 seconds (for test example 2970, see group "JPDDfirst") to almost 62 seconds (for test example 2902, see group "TMfirst"), which is 6.2 times more.

Looking at the descriptive data of the *differences* between JPDDs and Tracematches in Table 7.7 and Table 7.8, the eye gets caught by the fact that all medians of the differences in response time for all test examples of task T1 are negative, i.e. it seems that JPDDs always perform faster than Tracematches. The median varies from less than −117 seconds (for test example 4773, see group "TMfirst") to less than −2 seconds (for test example 5930, see group "JPDDfirst"). In contrast to that, there are both positive and negative medians for task T2, which renders this case likely to be undecided. The median varies

from more than −8 seconds (for test example 2876, see group "TMfirst") to almost +32 seconds (for test example 2902, see group "TMfirst").

When comparing the descriptive data between groups of participants rather than between notations, it can be observed that participants starting out with JPDDs ("JPDDfirst") often have quicker response times than participants beginning with Tracematches ("TMfirst"). This fact can be most easily observed in Table 7.9 and Table 7.11 which shows means and standard deviations in a transposed manner (in these tables, a greater-than/smaller-than sign in the center column indicates which one of the groups produced the greater means). Of course, it is also manifest in Table 7.7 and Table 7.8. Quicker response times are particularly produced for all questions of task T2 (see Table 7.8 and Table 7.11) and for those questions of task T1 relating to Tracematches (see left half of Table 7.7 and bottom half of Table 7.9). In case of the latter, participants starting out with JPDDs ("JPDDfirst") often even produce response times with a lower standard deviation. Response times are balanced between groups for questions of task T1 relating to JPDDs (see center columns of Table 7.7 and top half of Table 7.9).

Table 7.8 Descriptive data for task T2.

T2	TM		JPDD		Difference	
	median	sd	median	sd	median	sd
JPDDfirst (N=17)						
2182	21,368.0	5,569.90	27,034.0	20,014.15	6,563.0	18,710.38
2214	18,794.0	13,567.37	17,237.0	11,154.27	**-1,154.0**	16,153.72
2216	15,630.0	9,727.98	10,402.0	4,190.65	**-3,380.0**	9,613.76
2712	15,722.0	7,400.38	22,808.0	17,611.67	6,968.0	15,926.77
2770	23,070.0	10,944.61	25,697.0	16,762.45	4,000.0	16,822.13
2816	18,050.0	4,693.53	13,759.0	3,620.53	**-4,653.0**	5,711.94
2820	16,767.0	4,999.45	19,793.0	10,071.57	2,915.0	9,218.61
2876	18,217.0	4,271.80	16,941.0	6,856.70	**-897.0**	6,252.06
2902	25,800.0	8,493.13	45,892.0	78,665.96	20,931.0	73,694.77
2936	19,269.0	17,374.51	25,084.0	6,884.12	3,583.0	17,074.20
2970	**15,005.0**	4,249.93	**9,922.0**	4,238.53	**-6,089.0**	3,868.50
2988	16,522.0	6,410.30	10,936.0	3,550.14	**-4,255.0**	6,630.84
TMfirst (N=18)						
2182	19,705.5	6,071.16	30,136.5	36,425.16	8,181.0	36,666.74
2214	18,804.0	7,432.04	19,534.0	35,913.00	1,746.0	32,401.95
2216	20,686.5	5,261.60	15,669.0	5,337.40	**-5,702.5**	5,223.79
2712	23,749.5	8,202.47	43,394.5	17,850.11	18,228.5	14,885.22
2770	32,521.0	10,167.08	39,754.5	30,660.80	3,540.0	34,372.81
2816	18,236.0	9,354.13	15,684.0	5,084.48	**-2,105.0**	9,307.81
2820	22,106.0	6,373.55	18,669.5	5,240.94	**-2,763.0**	6,259.80
2876	28,540.5	11,271.47	20,189.0	9,158.78	**-7,815.5**	12,491.29
2902	**33,878.5**	12,094.84	**61,560.5**	30,042.99	31,862.0	31,883.06
2936	24,832.0	8,249.31	27,115.5	7,922.38	1,899.5	10,543.12
2970	15,730.5	4,568.01	11,729.5	4,218.53	**-3,597.5**	4,977.41
2988	18,251.5	5,376.67	12,539.0	3,505.01	**-4,845.0**	5,195.79

7.5.3 Hypotheses Testing

Following the test design, this section conducts a three-way ANOVA with repeated measurements on two variables. The within-subject variables are "notation" and "test example". The between-subject variable is "alternation". At first, the section evaluates if the measured data satisfies the assumptions of an ANOVA using the Shapiro-Wilk test on

Table 7.9 Shapiro-Wilk test on normality and Levene test on homoscedasticy for task T1.

T1	JPDDfirst (N=16)				TMfirst (N=15)			
	mean	sd	shapiro		mean	sd	shapiro	levene
JPDD								
4705	13,518.50	4,508.36	0.039	<	15,011.53	8,670.19	0.000	**0.188**
4709	19,662.63	23,312.49	0.000	>	13,959.07	4,613.79	0.016	**0.151**
4715	9,471.31	4,186.79	0.024	>	9,088.93	2,761.87	0.036	**0.340**
4731	10,999.63	2,925.23	0.036	>	10,410.13	2,693.26	**0.434**	**0.969**
4741	7,540.19	2,186.01	**0.083**	<	7,725.00	2,882.13	0.002	**0.412**
4751	9,671.06	3,416.25	0.006	>	9,184.33	2,785.56	**0.074**	**0.922**
4755	11,738.25	2,242.95	**0.796**	<	12,040.80	2,721.58	**0.334**	**0.273**
4761	**6,241.69**	1,416.58	0.049	<	7,329.53	2,853.24	**0.202**	0.008
4767	13,807.81	7,668.81	0.000	>	12,076.93	4,918.96	0.000	**0.331**
4773	14,749.94	5,224.36	0.002	<	15,089.07	5,421.25	**0.369**	**0.427**
5756	12,176.00	5,520.80	0.011	<	13,172.87	7,698.75	0.001	**0.464**
5772	7,957.19	3,077.92	0.000	<	8,284.87	3,496.89	0.004	**0.624**
5930	7,346.19	1,646.94	**0.527**	>	**6,865.87**	2,398.35	0.002	**0.461**
5936	12,975.38	13,163.67	0.000	>	10,529.20	4,076.73	0.009	**0.343**
5988	13,288.50	6,916.65	0.001	>	12,751.80	5,564.37	**0.225**	**0.675**
6022	11,201.44	2,112.48	**0.414**	<	12,227.13	3,471.91	**0.143**	**0.086**
TM								
4705	61,839.63	39,737.15	0.003	<	70,515.07	37,719.37	**0.091**	**0.937**
4709	65,122.31	35,843.41	0.045	<	98,240.53	64,053.72	0.026	**0.094**
4715	11,224.81	3,225.76	**0.450**	<	21,122.27	14,437.40	0.021	0.001
4731	37,796.13	31,354.24	0.002	<	46,585.67	20,908.07	**0.763**	**0.307**
4741	13,220.06	6,598.82	**0.137**	<	21,231.60	13,269.16	**0.214**	0.002
4751	13,757.13	5,280.52	**0.226**	<	27,601.80	13,413.31	**0.346**	0.037
4755	26,660.19	15,687.96	0.051	<	33,308.27	13,824.20	**0.597**	**0.555**
4761	13,159.38	6,125.22	**0.336**	<	16,871.53	13,651.77	0.002	0.041
4767	45,179.94	32,510.55	0.006	<	73,890.87	65,250.31	0.000	**0.275**
4773	**101,050.81**	60,380.24	**0.108**	<	**140,194.00**	92,296.74	**0.099**	0.037
5756	44,974.88	27,807.28	0.013	<	57,647.20	33,056.60	0.051	**0.358**
5772	22,145.63	8,757.46	**0.579**	<	34,126.87	17,482.82	**0.171**	0.002
5930	13,316.38	7,467.50	0.007	<	25,856.00	19,715.32	0.039	0.010
5936	74,087.81	66,499.38	0.000	<	62,293.00	38,563.50	0.029	**0.413**
5988	26,166.19	14,939.35	0.022	<	42,061.13	21,295.62	**0.874**	**0.287**
6022	40,569.81	28,853.29	0.003	<	54,850.27	29,281.13	0.004	**0.555**

Table 7.10 Mauchly test on sphericity for task T1.

T1 inner subject effects	Mauchly W	approx. χ^2	df	p	Greenhouse-Geisser ε	Huynh-Feldt ε	lower bound ε
testexample	0.000	522.266	119	**0.000**	0.304	0.381	0.067
notation	1.000	0.000	0	**.**	1.000	1.000	1.000
testexample*notation	0.000	539.550	119	**0.000**	0.316	0.398	0.067

Table 7.11 Shapiro-Wilk test on normality and Levene test on homoscedasticy for task T2.

T2	JPDDfirst (N=17)					TMfirst (N=18)			
	mean	sd	shapiro			mean	sd	shapiro	levene
JPDD									
2182	34,025.41	20,014.15	0.004	<		41,899.11	36,425.16	0.000	**0.430**
2214	21,777.18	11,154.27	0.000	<		31,340.06	35,913.00	0.000	**0.109**
2216	12,608.06	4,190.65	0.002	<		15,107.67	5,337.40	**0.279**	**0.152**
2712	29,845.06	17,611.67	0.035	<		45,712.39	17,850.11	0.004	**0.545**
2770	31,565.41	16,762.45	0.001	<		46,254.33	30,660.80	0.006	0.055
2816	13,727.65	3,620.53	**0.127**	<		16,886.39	5,084.48	**0.132**	**0.393**
2820	21,661.35	10,071.57	0.001	>		18,924.78	5,240.94	**0.989**	**0.281**
2876	18,477.00	6,856.70	0.025	<		22,478.17	9,158.78	**0.356**	**0.294**
2902	**74,376.24**	78,665.96	0.000	>		**68,450.67**	30,042.99	0.051	**0.211**
2936	23,792.82	6,884.12	**0.602**	<		29,018.11	7,922.38	**0.064**	**0.697**
2970	**10,757.06**	4,238.53	0.000	<		**12,565.61**	4,218.53	**0.171**	**0.273**
2988	11,537.53	3,550.14	**0.219**	<		12,786.72	3,505.01	**0.151**	**0.741**
TM									
2182	22,070.47	5,569.90	**0.136**	>		21,337.83	6,071.16	**0.310**	**0.630**
2214	25,878.00	13,567.37	0.009	>		20,443.33	7,432.04	**0.389**	0.017
2216	18,272.76	9,727.98	0.000	<		20,559.83	5,261.60	**0.312**	**0.310**
2712	17,687.59	7,400.38	**0.093**	<		24,390.44	8,202.47	**0.792**	**0.659**
2770	26,113.82	10,944.61	0.029	<		33,570.17	10,167.08	**0.376**	**0.998**
2816	18,543.53	4,693.53	0.075	<		20,719.28	9,354.13	0.001	**0.119**
2820	16,087.12	4,999.45	**0.117**	<		22,562.89	6,373.55	**0.169**	**0.368**
2876	19,138.65	4,271.80	**0.239**	<		31,065.33	11,271.47	0.016	0.022
2902	26,523.00	8,493.13	**0.164**	<		34,365.89	12,094.84	**0.060**	**0.381**
2936	25,930.41	17,374.51	0.000	<		26,221.67	8,249.31	0.050	**0.107**
2970	15,985.24	4,249.93	0.072	>		15,877.83	4,568.01	0.000	**0.497**
2988	17,272.12	6,410.30	0.001	<		19,004.11	5,376.67	**0.302**	**0.812**

Table 7.12 Mauchly test on sphericity for task T2.

T2 inner subject effects	Mauchly W	approx. χ^2	df	p	Greenhouse-Geisser ε	Huynh-Feldt ε	lower bound ε
testexample	0.000	392.596	65	**0.000**	0.259	0.295	0.091
notation	1.000	0.000	0	**.**	1.000	1.000	1.000
testexample*notation	0.000	402.347	65	**0.000**	0.272	0.311	0.091

normality [*Shapiro & Wilk (1965)*], the Levene test on homoscedasticy [*Levene (1960)*], and the Mauchly test on sphericity [*Mauchly (1940)*]. Subsequently, the section performs the ANOVA and reports on significant variables, significant interactions of such variables, and interesting confidence intervals.

Table 7.9 and Table 7.11 give the p-values of the Shapiro-Wilk test and of the Levene test on the measured response times, divided by alternation ({"JPDDfirst", "TMfirst"}), notation ({"JPDD", "TM"}), and task ({"T1", "T2"}). The Shapiro-Wilk test evaluates if the measured response times are normally distributed (a value<0.05 indicates that the normality assumption must be rejected, otherwise it may be maintained). The Levene test evaluates if the variance of the response times is homogeneous across groups of participants for a given test example and a given notation (again, a value<0.05 indicates that the homogeneity assumption must be rejected, otherwise it may be maintained). Table

7.9 and Table 7.11 highlight all values according to which the corresponding assumption may be maintained using a bold font. Accordingly, the normality assumption must be rejected in most cases, whereas the homogeneity assumption may be maintained for most cases.

Table 7.10 and Table 7.12 show the results of the Mauchly test for sphericity, which tests if the variance of the *differences* between the response times is homogenous for all pairs of treatments of the within-subject variables and of their interactions with the other variables (a p-value<0.05 indicates that the sphericity assumption must be rejected, otherwise it may be maintained). Table 7.10 and Table 7.12 show that the sphericity assumption is fulfilled for variable "notation" (which is no surprise because there is only one pair of treatments, i.e. {"JPDD","TM"}). The sphericity assumption must be rejected, though, for variable "test example", as well as for the interaction effect between "test example" and "notation" (since p=0.000[58]<0.05).

In conclusion, the measured response times do not satisfy all of the assumptions of an ANOVA, unfortunately. This means that the actual α-error and the actual power of the ANOVA may differ from the nominal ones if an ANOVA is performed anyway. Fortunately, (α-error and power of) an ANOVA have proven to be quite robust against violations of its normality assumption and its homogeneity assumption (cf. [*Gene V. Glass & Sanders (1972)*]) – in particular if the number of test participants in each group is reasonably large and equal in size. Considering that the groups of this test are reasonably large (N(group)>10) and almost of same size (i.e. N("JPDDfirst")=16 and N("TMfirst")=15 for task T1, and N("JPDDfirst")=17 and N("TMfirst")=18 for task T2), performing an ANOVA seems justifiable despite of the violations of its normality assumption and of its homogeneity assumption.

However, the violation of the sphericity assumptions needs to be taken into account (by a reduction of the degrees of freedom (df)) for variable "test example" and for its interaction effect with variable "notation". To do so, the approaches of Box [*Box (1954)*] and of Greenhouse and Geisser [*Geisser & Greenhouse (1958)*] are chosen. These approaches are chosen, rather than another approach of Huynh and Feldt [*Huynh & Feldt (1976)*], because sphericity is heavily violated (i.e. ε<0.75; see Table 7.10 and Table 7.12).

The results of the ANOVA are shown in Table 7.13, Table 7.14, Table 7.16, and Table 7.17. An ANOVA tests if the marginal means of the measured data is the same for all pairs of treatments of a given independent variable. Additionally, it checks if there is no extra (alleviating or reinforcing) effect resulting from a combination of the treatments of two (or more) independent variables, i.e. if there is no "interaction effect" between variables. If the former assumption is rejected, there must be a significant difference between at least one pair of treatments of the respective variable. If the latter assumption is rejected, there must be a significant difference between the estimated, i.e. purely additive, data and the measured data for at least one combination of the treatments of the given two (or more) variables. In consequence, the variable – or its interaction with another variable – is considered to have a significant effect on the measured data.

The p-values shown in Table 7.14 and Table 7.17 indicate that both within-subject variables, i.e. "notation" as well as "test example", have a strong significant effect on the

[58] Note that none of the p-values mentioned in this thesis is literally 0; rather, a p-value of 0.000 indicates that the true p-value is smaller than 10^{-3}.

Table 7.13 Test of between-subject effects for task T1.

T1			p	
alternation			**0.044**	*

Table 7.14 Test of within-subject effects for task T1.

T1		df	p	
notation	sphericity given	1	**0.000**	**
notation * alternation	sphericity given	1	**0.012**	*
testexample	sphericity assumed	15	0.000	**
	Greenhouse-Geisser	4.566	**0.000**	**
	lower bound	1.000	**0.000**	**
testexample * alternation	sphericity assumed	15	0.485	
	Greenhouse-Geisser	4.566	0.433	
	lower bound	1.000	0.333	
testexample * notation	sphericity assumed	15	0.000	**
	Greenhouse-Geisser	4.741	**0.000**	**
	lower bound	1.000	**0.000**	**
testexample * notation * alternation	sphericity assumed	15	0.328	
	Greenhouse-Geisser	4.741	0.348	
	lower bound	1.000	0.297	

Table 7.15 Confidence intervals of dichotomous variables "alternation" and "notation" for task T1.

T1					confidence interval (99%)	
variable	difference	p	marg. mean	std. error	l. bound	u. bound
alternation	"JPDDfirst" vs. "TMfirst"	0.044	-6,547.70	3,102.68	**-15,099.89**	**2,004.49**
notation	"JPDD" vs. "TM"	0.000	-33,705.45	2,598.83	**-40,868.82**	**-26,542.08**

response times of both tasks (since $p=0.000<\alpha=0.01$ or $p=0.002<\alpha=0.01$, respectively) – and so does their interaction "test example * notation" (since $p=0.000<\alpha=0.01$ and $p=0.001<\alpha=0.01$, respectively). Note that in all three cases, the strong significance even shows at the most conservative correction of the sphericity violation (using "lower bounds", which actually assumes maximal violation of the sphericity assumptions). Furthermore, Table 7.13 and Table 7.16 indicate that there are close-to-significant differences between the different treatments of the between-subject variable "alternation" ({"TMfirst", "JPDDfirst"}) for both tasks (since $p=0.044<\alpha=0.10$ and $p=0.081<\alpha=0.10$, respectively). Likewise, Table 7.14 reports that there is a close-to-significant impact of the interaction between "notation" and "alternation" in task T1 (since $p=0.012<\alpha=0.10$). All other interactions are insignificant (since $p\geq\alpha=0.10$).

Table 7.15 and Table 7.18 show the confidence intervals of the means of the differences between the different treatments of the dichotomous variables "alternation" and "notation". In compliance with the calculated p-values, the confidence interval of the variable "alternation" contains 0 for both tasks T1 and T2 (denoting that there is no significant difference between the different treatments of the variable), whereas the confidence interval of the variable "notation" is completely below 0 for task T1 and over 0 for task T2 (which means that there is a significant difference between the different

Table 7.16 Test of between-subject effects for task T2.

T2			p	
alternation			**0.081**	+

Table 7.17 Test of within-subject effects for task T2.

T2		df	p	
notation	sphericity given	1	**0.002**	**
notation * alternation	sphericity given	1	0.656	
testexample	sphericity assumed	11	0.000	**
	Greenhouse-Geisser	2.854	**0.000**	**
	lower bound	1.000	**0.000**	**
testexample * alternation	sphericity assumed	11	0.446	
	Greenhouse-Geisser	2.854	0.394	
	lower bound	1.000	0.325	
testexample * notation	sphericity assumed	11	0.000	**
	Greenhouse-Geisser	2.992	**0.000**	**
	lower bound	1.000	**0.001**	**
testexample * notation * alternation	sphericity assumed	11	0.277	
	Greenhouse-Geisser	2.992	0.309	
	lower bound	1.000	0.279	

Table 7.18 Confidence intervals of dichotomous variables "alternation" and "notation" for task T2.

T2					confidence interval (99%)	
variable	difference	p	marg. mean	std. error	l. bound	u. bound
alternation	"JPDDfirst" vs. "TMfirst"	0.081	-4,078.71	2,262.20	**-10,261.94**	**2,104.51**
notation	"JPDD" vs. "TM"	0.002	5,248.06	1,543.58	**1,029.04**	**9,467.08**

treatments of the variable). Furthermore, the confidence intervals of the variable "notation" indicate that response times related to JPDDs may be about 27 to 41 seconds faster than the response times related to Tracematches in case of task T1. Whereas in case of task T2, response times related to JPDDs may be about 1 to 9 seconds slower than the response times related to Tracematches. (For the sake of completeness, note that the confidence intervals of the variable "alternation" reveals that the response times of group "JPDDfirst" may differ from the response times of group "TMfirst" from about −15 to about +2 seconds in case of task T1, while in case of task T2 they may differ from about −10 to about +2 seconds.) The differences between the polytomous variable "test example" will be further investigated in the next section 7.5.4.

Having identified a significant interaction effect between the independent variables, it is necessary to look at the marginal means of the treatments of these variables in order to qualify the main effects of those variables. Table 7.19, Table 7.20, and Table 7.21 report on the marginal means of these interaction effects as well as on their confidence intervals. Figure 7.3, Figure 7.4, and Figure 7.5 visualize these marginal means and their confidence intervals with the help of interaction diagrams.

Figure 7.3 reveals that the interaction between "notation" and "test example" is of hybrid nature in case of task T1. This means that the marginal means of the treatments of

the variable "notation" maintain the same relationship (i.e. TM > JPDD) for all treatments of the variable "test example", i.e. all marginal means relating to Tracematches are higher than the corresponding marginal means relating to JPDDs (see top chart in Figure 7.3). However, the reverse is not true: the marginal means of the treatments of variable "test example" do not maintain the same order for both treatments of the variable "notation". That is, there are pairs of test examples where one leads to quicker response times when shown as a Tracematch, while the other leads to quicker response times when presented as a JPDD (examples are test examples 4705 and 5936, or 4731 and 5988, or 4751 and 5772, or 4715 and 5930; see bottom chart in Figure 7.3[59]). If variables are involved in hybrid interactions, only the variable whose marginal means maintain a uniform trend for all treatments of the other variable should be interpreted in isolation.

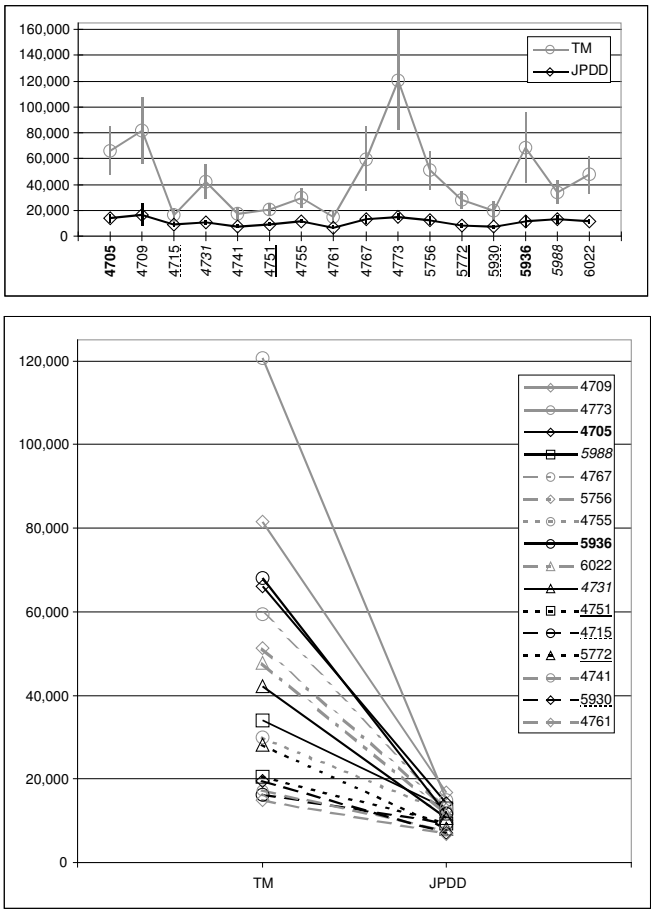


Figure 7.3 Interaction diagrams for significant interaction effects found in task T1.

Figure 7.4 investigates the same interaction between "notation" and "text example" for task T2[60]. This time, none of the interacting variables maintains the same relationship between the marginal means of all of its treatments for every treatment of the other variable (see test examples 2712 and 2876, for example, whose marginal means change order depending on the treatments of both variables, i.e. one test example produces the

[59] Note that the test examples in the legend of the bottom chart of Figure 7.3 are ordered according to the marginal means observed for JPDDs.

[60] Note that the test examples in the legend of the bottom chart of Figure 7.4 are ordered according to the marginal means observed for Tracematches.

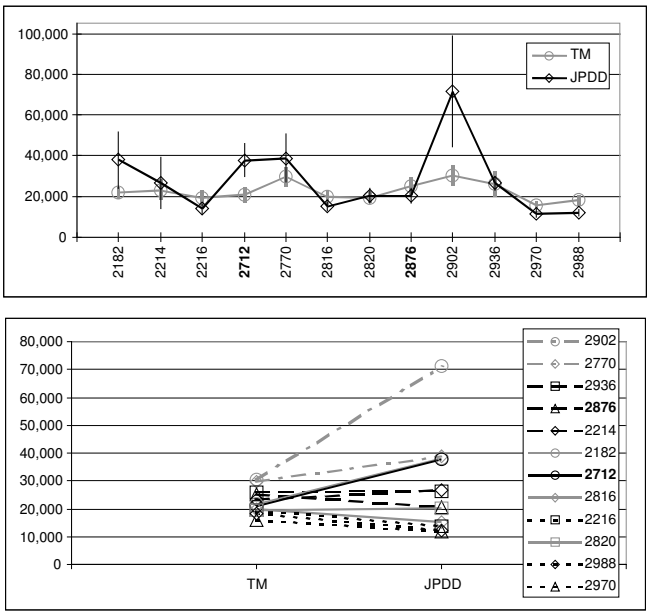


Figure 7.4 Interaction diagrams for significant interaction effects found in task T2.

higher marginal means for one treatment, while the other test example produces the higher marginal means for the other treatment of a variable). In consequence, this interaction can be classified as disordinal. Variables which are involved in disordinal interactions should always be interpreted in combination rather than in isolation.

Finally, Figure 7.5 investigates the close-to-significant interaction effect between the notation and the alternation in task T1. Again there is a hybrid interaction: the order of the marginal means of the different treatments of the notation is the same for both alternations, i.e. the marginal means relating to Tracematches are always higher than the corresponding marginal means relating to JPDDs (see left chart of Figure 7.5). Yet, the order of the marginal means of the alternations differs for each treatment of the notation, i.e. the marginal mean of group "TMfirst" is higher than the marginal mean of group "JPDDfirst" for Tracematches, whereas the marginal mean of group "JPDDfirst" is higher than the marginal mean of group "TMfirst" for JPDDs (see right chart in Figure 7.5). In consequence, the close-to-significant effect of the alternation should not be interpreted in isolation in case of task T1 (even though the difference between the marginal means of the different groups is not very big for JPDDs, i.e. roughly 11 seconds for group "TMfirst" vs. approx. 11.4 seconds for group "JPDDfirst"; see Table 7.20).

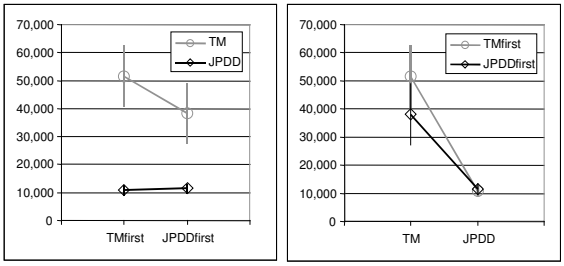


Figure 7.5 Interaction diagrams for more significant interaction effects found in task T1.

In summary, the ANOVA reveals a significant effect of the notation for both tasks T1 and T2 (i.e. $p=0.000<\alpha=0.01$ for task T1 and $p=0.002<\alpha=0.01$ for task T2, respectively).

At the same time, test examples happen to have a significant effect on the response times, too (i.e. $p=0.000<\alpha=0.01$ for both tasks T1 and T2). For task T2, the notation interacts with the test examples in a significant (i.e. $p=0.001<\alpha=0.01$) and disordinal way, meaning that the significant impact of the notation depends on the test example one is looking at. For task T1, the interaction between notation and test examples is significant, too (i.e. $p=0.000<\alpha=0.01$), yet it is hybrid, meaning that it does not overrule or alter the central tendency of the impact of the notation on the response times of each test example.

The confidence interval of the means of the differences in response time between the different notations measured for task T1 indicate that JPDDs may lead to a decrease in response times of about 27 to 41 seconds in comparison to Tracematches in case of task T1. In case of task T2, the confidence interval indicates a possible increase in response times for JPDDs of about 1 to 9 seconds in comparison to Tracematches. This increase in response times should not be interpreted, though, due to the disordinal nature of the interaction between the notation and test examples in case of task T2.

Instead, one may take a look at the confidence intervals relating to the individual test examples (see Table 7.21) in order to find out that – depending on the test examples – the confidence intervals of JPDDs may be below than the confidence intervals of the Tracematches (such as in case of test example 2988 and 2970), or the confidence intervals of Tracematches may be below than the confidence intervals of the JPDDs (such as in case of test example 2712 and 2902), or the confidence intervals of JPDDs and Tracematches may be overlapping (such as in case of test example 2770 or 2876), or they may be even nested (such as in case of test example 2214 or 2936). Note that this is in contrast to the individual confidence intervals of the test examples of task T1 (see Table 7.19), where the confidence intervals of JPDDs are always below the confidence intervals of Tracematches.

Furthermore, the alternation of notations used to show the test examples to the participants turns out to have a close-to-significant impact on the measured test results (i.e. $p=0.044<\alpha=0.10$ for task T1 and $p=0.081<\alpha=0.10$ for task T2, respectively). Likewise, there is a close-to-significant (i.e. $p=0.012<\alpha=0.10$) and hybrid interaction effect between alternation and notation in case of task T1, which means that – at an α-level of 0.10 – the impact of the alternation effect would depend on the notation (while the impact of the notation would be independent from the alternation).

The corresponding confidence intervals of the means of the differences in response time between the different alternation of notations indicate that alternation "JPDDfirst" may lead to a difference from about −15 to about +2 seconds in comparison to alternation "TMfirst" in case of task T1, while in case of task T2 the differences may range from about −10 to about +2 seconds (note that due to the close-to-significant interaction between alternation and notation in case of task T1, the confidence interval of the differences in response time between the different alternation of notations should not be interpreted in isolation in case of task T1).

7.5.4 Data Exploration

After having completed the statistical hypothesis testing (with the conclusion that JPDDs are indeed capable of facilitating the detection and comprehension of data constraints in join point selections in comparison to Tracematches), the goal of this section is to take a closer look at the details of the measured data. To do so, the overall

*Table 7.19 Estimated marginal means of interaction ("notation" * "test example") for task T1.*

T1	TM		confidence interval (99%)		JPDD		confidence interval (99%)	
	marg. mean	std. error	l. bound	u. bound	marg. mean	std. error	l. bound	u. bound
4705	66,177.35	6,968.03	46,970.77	85,383.93	14,265.02	1,229.37	10,876.40	17,653.63
4709	81,681.42	9,242.23	56,206.28	107,156.57	16,810.85	3,067.45	8,355.77	25,265.92
4715	16,173.54	1,850.18	11,073.73	21,273.35	9,280.12	641.64	7,511.53	11,048.72
4731	42,190.90	4,820.25	28,904.42	55,477.37	10,704.88	505.97	9,310.24	12,099.52
4741	17,225.83	1,863.35	12,089.71	22,361.95	7,632.59	457.50	6,371.54	8,893.65
4751	20,679.46	1,808.45	15,694.69	25,664.23	9,427.70	562.04	7,878.49	10,976.91
4755	29,984.23	2,662.69	22,644.83	37,323.63	11,889.53	446.65	10,658.39	13,120.67
4761	15,015.45	1,879.36	9,835.21	20,195.70	6,785.61	400.53	5,681.58	7,889.64
4767	59,535.40	9,166.55	34,268.85	84,801.96	12,942.37	1,165.97	9,728.51	16,156.24
4773	120,622.41	13,917.34	82,260.85	158,983.97	14,919.50	956.06	12,284.24	17,554.76
5756	51,311.04	5,472.66	36,226.28	66,395.79	12,674.43	1,197.10	9,374.75	15,974.11
5772	28,136.25	2,458.81	21,358.81	34,913.68	8,121.03	590.65	6,492.98	9,749.07
5930	19,586.19	2,644.01	12,298.28	26,874.09	7,106.03	367.39	6,093.36	8,118.69
5936	68,190.41	9,851.13	41,036.88	95,343.93	11,752.29	1,775.77	6,857.59	16,646.99
5988	34,113.66	3,285.95	25,056.32	43,171.00	13,020.15	1,132.13	9,899.55	16,140.75
6022	47,710.04	5,222.16	33,315.76	62,104.32	11,714.29	512.30	10,302.19	13,126.38

*Table 7.20 Estimated marginal means of interaction ("notation" * "alternation") for task T1.*

T1	TM		confidence interval (99%)		JPDD		confidence interval (99%)	
	marg. mean	std. error	l. bound	u. bound	marg. mean	std. error	l. bound	u. bound
TMfirst	51,649.75	4,044.90	40,500.45	62,799.06	10,984.19	740.07	8,944.28	13,024.10
JPDDfirst	38,141.94	3,916.46	27,346.67	48,937.21	11,396.61	716.57	9,421.47	13,371.74

*Table 7.21 Estimated marginal means of interaction ("notation" * "test example") for task T2.*

T2	TM		confidence interval (99%)		JPDD		confidence interval (99%)	
	marg. mean	std. error	l. bound	u. bound	marg. mean	std. error	l. bound	u. bound
2182	21,704.15	986.45	19,007.92	24,400.38	37,962.26	5,009.79	24,269.12	51,655.41
2214	23,160.67	1,834.58	18,146.26	28,175.07	26,558.62	4,552.33	14,115.84	39,001.39
2216	19,416.30	1,311.42	15,831.83	23,000.77	13,857.86	814.32	11,632.10	16,083.63
2712	21,039.02	1,323.01	17,422.86	24,655.18	37,778.72	2,998.97	29,581.71	45,975.74
2770	29,842.00	1,784.21	24,965.26	34,718.74	38,909.87	4,212.31	27,396.45	50,423.29
2816	19,631.40	1,262.67	16,180.17	23,082.64	15,307.02	750.03	13,256.97	17,357.07
2820	19,325.00	972.07	16,668.07	21,981.94	20,293.07	1,345.71	16,614.86	23,971.27
2876	25,101.99	1,457.56	21,118.09	29,085.89	20,477.58	1,373.85	16,722.47	24,232.70
2902	30,444.44	1,776.22	25,589.56	35,299.33	71,413.45	9,954.47	44,205.13	98,621.77
2936	26,076.04	2,277.64	19,850.61	32,301.47	26,405.47	1,257.61	22,968.06	29,842.88
2970	15,931.53	746.86	13,890.17	17,972.90	11,661.34	715.00	9,707.06	13,615.61
2988	18,138.11	997.77	15,410.93	20,865.30	12,162.13	596.41	10,531.97	13,792.28

marginal means of the response times are split into more specific marginal means for each (combination of) notation and alternation. This is motivated by the fact that both notation and alternation have shown to have a significant impact, or a close-to-significant impact, respectively, on the response times. Therefore, it is interesting to see how the response times of each notation and each alternation vary during the test.

One interesting thing to explore is to investigate if the response times of the notations and the alternations depend on any particular characteristic of the test examples. In order to do so, the marginal means of the measured response times of each of the test examples are ordered according to each of the varying characteristics of the test examples presented in Table 7.3 and Table 7.4. And indeed, if the test examples of task T1 are ordered according to the number of answer nodes, a systematic (even though not strict) increase of the marginal means of the response times becomes visible (see Figure 7.6). And in case of task T2, a likewise systematic (yet not strict) increase in the marginal means of the response times becomes visible if the test examples are ordered according to the number of question nodes (see Figure 7.7).

In both cases there is a noteworthy decrease in response time for the test examples with the highest number of answer nodes and question nodes, though, i.e. for test examples 4705 and 6022 in case of task T1 (see Figure 7.6) and for test example 2712 in case of task T2 (see Figure 7.7).

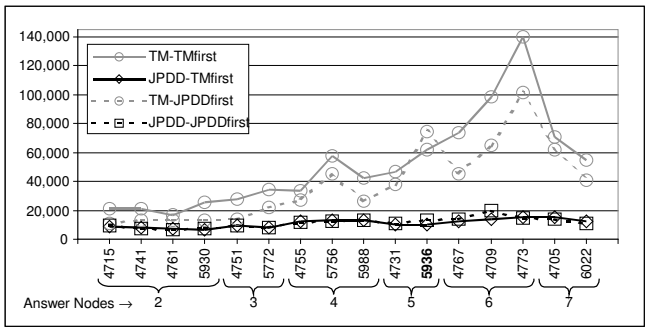


Figure 7.6 Marginal means of response times of task T1 divided by notation and alternation and ordered by number of answer nodes.

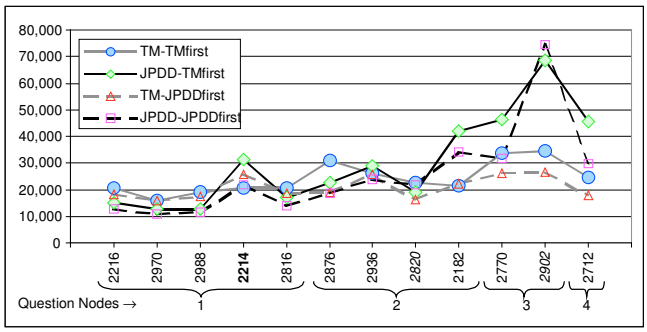


Figure 7.7 Marginal means of response times of task T2 divided by notation and alternation and ordered by number of question nodes.

Another noteworthy observation about task T1 (see Figure 7.6) is that response times remain fairly stable for JPDDs across test examples, whereas response times for Tracematches vary considerably. As a result from this fact, the variation of the *differences* between the notations almost only arises from the variation of the response times for

Tracematches (see Figure 7.8). Accordingly, the course of the marginal means of the differences in response time between JPDDs and Tracematches shown in Figure 7.8 looks almost like a mirror image of the course of the marginal means of the response times for Tracematches shown in Figure 7.6.

Moreover, it can be observed that response times of JPDDs are alike for both alternations in case of task T1 (see Figure 7.6), whereas response times of Tracematches differ notably between alternations, i.e. participants starting out with JPDDs almost always produce quicker response times than participants starting out with Tracematches (there is only one exception for test example 5936). In consequence, the attained benefit in response time when using JPDDs (as opposed to Tracematches) is usually smaller for participants of group "JPDDfirst" than for participants of group "TMfirst" (see Figure 7.8).

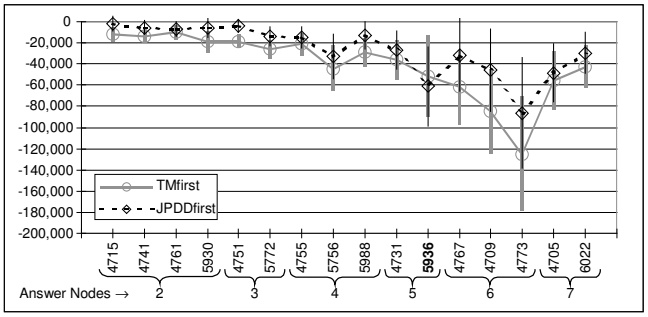


Figure 7.8 Marginal means of the differences in response time of task T1 between notations divided by alternation and ordered by number of answer nodes.

In case of task T2, it can be observed that response times vary notably with the treatments of both variables, i.e. with notation and alternation. Yet again, participants starting out with JPDDs (see dashed lines with empty triangles and squares in Figure 7.7) almost always produce quicker response times than participants starting out with Tracematches (see solid lines with filled circles and diamonds in Figure 7.7). This time this is true for both notations. There are exceptions, though (see test example 2214 for Tracematches, represented by gray lines with circles and triangles in Figure 7.7, and test examples 2820 and 2902 for JPDDs, represented by black lines with diamonds and squares in Figure 7.7).

As a result, the differences in response time between the different notations vary with the treatments of both variables in case of task T2. Figure 7.9 shows the marginal means of the differences in response time divided by alternation. As it turns out, the increase of the

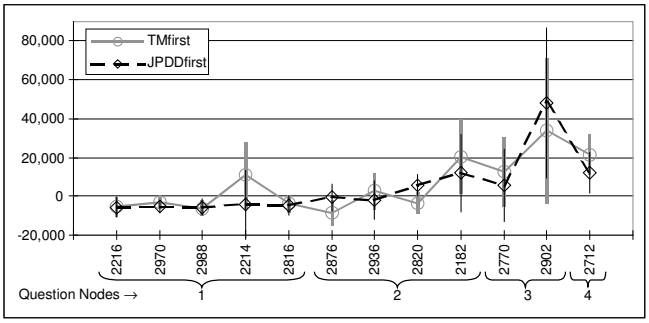


Figure 7.9 Marginal means of the differences in response time of task T2 between notations divided by alternation and ordered by number of question nodes.

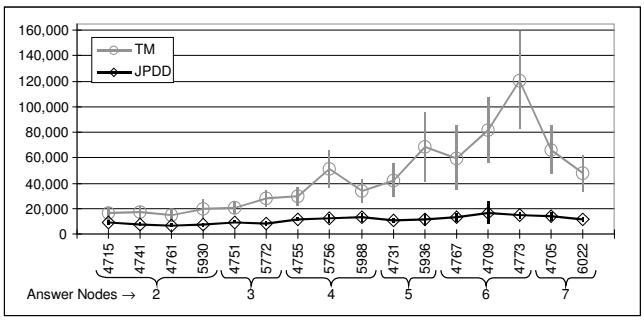


Figure 7.10 Marginal means of the response time of task T1 divided by notation and ordered by number of answer nodes.

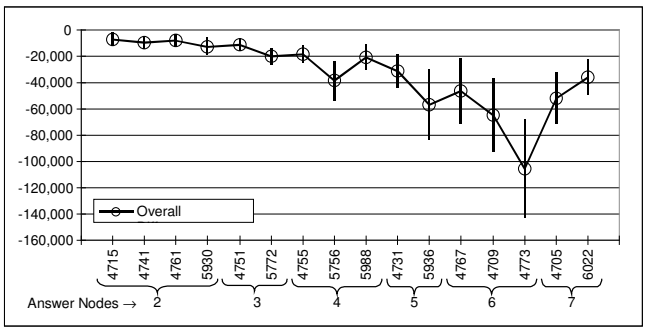


Figure 7.11 Marginal means of the differences in response time of task T1 between notations ordered by number of answer nodes.

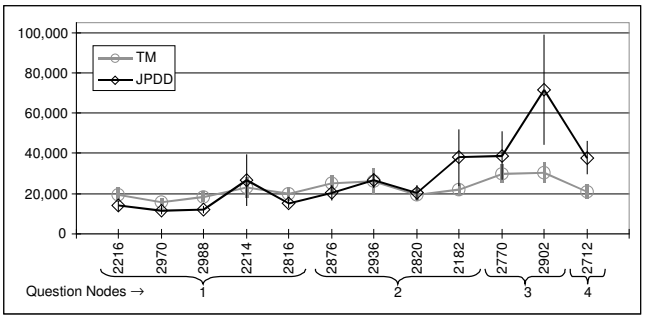


Figure 7.12 Marginal means of the response time of task T2 devided by notation and ordered by number of question nodes.

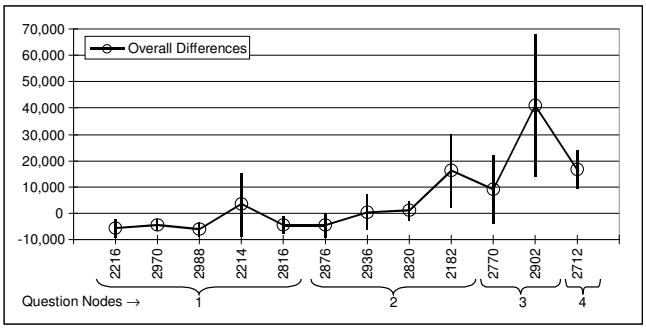


Figure 7.13 Marginal means of the differences in response time of task T2 between notations ordered by number of question nodes.

marginal means is remarkably steady for the group of participants starting out with JPDDs (at least up to the last three test examples with three and four question nodes). Furthermore, it suggests that there is a turning point where the response times of Tracematches begin to be shorter than the response times of JPDDs (i.e. the measured response times pass the base line when test examples have two question nodes). In contrast to that, the increase of response times is more erratic for the group of participants starting out with Tracematches.

In conclusion, the variation of the overall marginal means of the differences in response time between the different notations (and across groups; as shown in Figure 7.11) is primarily due to the variation of the marginal means of the response times for Tracematches in case of task T1 (as shown in Figure 7.10; note that Figure 7.10 depicts the same information as the top chart of Figure 7.3, yet in a different order).

In contrast to that, the overall marginal means of the differences in response time between the different notations (and across groups; as shown in Figure 7.12) is dominated by the erratic course of the response times related to JPDDs which are produced by the group of participants starting out with Tracematches, and thus follows a more erratic course (as shown by Figure 7.13).

7.6 Conclusion and Interpretation

7.6.1 Summary

The data analysis in the previous section shows that the null hypothesis, which claims that there is *no* difference between JPDDs and Tracematches with respect to the comprehensibility of data constraints in join point selections, can be clearly rejected.

For both tasks T1 and T2, a significant difference can be measured at an α-level of 0.01 (see Table 7.14 and Table 7.17). However, in both cases the difference in response time also significantly depends on the test examples (see same tables). While for task T1, JPDDs still always perform better than Tracematches – even though with different magnitudes (see Figure 7.11) – there is a turning point in case of task T2 at which Tracematches start to perform better than JPDDs (see Figure 7.13). According to Figure 7.13, this turning point seems to occur when participants have to perform variant `n=2` of task T2. The magnitude of the difference in case of task T1 seems to depend on the number of nodes that need to be selected in order to solve a task, whereas the turning point at which Tracematches start to perform better than JPDDs seems to depend on the variant of the task T2, i.e. the number of objects (o_1, ..., o_n with n equals 2, 3, or 4) given in the question.

In more concrete terms, this means that JPDDs may decrease response times from roughly 27 to almost 41 seconds in comparison to Tracematches in case of task T1 (as indicated by the overall confidence interval shown in Table 7.15), which denotes a decrease in response time of almost 74% to 76% in comparison to Tracematches (as revealed by Table C.2 in Appendix C) In contrast to that, JPDDs may both decrease response times by almost 9 seconds as well as increase response times by almost 68 seconds depending on the test example one is working on in case of task T2 (as indicated by the test example-specific confidence intervals shown in Table C.9 in Appendix C). This equates to a decrease in response time of roughly 20% (in case of test example 2876) and an increase in response

time of more than 64% (in case of test example 2902), respectively (as revealed by Table 7.21).

Apart from that, the alternation of notations in which the test examples are shown to the participants turns out to have a close-to-significant impact on the measured response times (i.e. alternation would have a significant impact at an α-level of 0.10; see Table 7.13 and Table 7.16): participants starting out with JPDDs repeatedly perform better than participants beginning with Tracematches (see Figure 7.6 and Figure 7.7). Interestingly, in case of task T1, the better performance of the former participants only shows for questions relating to JPDDs, while no such difference shows for questions relating to Tracematches (see Figure 7.6). This phenomenon is likely to explain the close-to-significant interaction effect between alternation and notation in case of task T1 (i.e. $p=0.012<\alpha=0.10$; see Table 7.14). In case of task T2, a slightly better performance of participants starting out with JPDDs shows for both notations (see Figure 7.7). A consequence of this may be that no significant or close-to-significant interaction effect between alternation and notation can be observed (i.e. $p=0.656\geq\alpha=0.10$; see Table 7.17).

7.6.2 Conclusions

One interesting observation about the experiment results is that the advantages of JPDDs over Tracematches apparently depend on the task which needs to be solved. This becomes manifest in the different results for task T1 and T2 (indicated by the confidence intervals shown in Table 7.15 and Table 7.18, one of which is completely negative, i.e. the one of task T1, and the other of which is completely positive, i.e. the one of task T2). Moreover, advantages even depend on variants of those tasks. This becomes manifest in the different results for the different variants of task T2 (revealed by the disordinal interaction effect between notation and test example shown in Figure 7.4, which means that the aforementioned confidence interval of task T2 should not be interpreted in isolation). In summary, there are tasks/variants where JPDDs are superior to Tracematches, and there are tasks/variants where Tracematches are more helpful than JPDDs. To be more concrete, JPDDs seem to help with the identification of dependent objects (i.e. of objects which are involved in more than two methods; cf. research questions Q1 and Q2), whereas Tracematches seem to be helpful when a set of methods needs to be identified which collectively involve multiple dependent objects (cf. research question Q5).

Another interesting observation is the positive relation between the (differences in) response time and the number of answers nodes in case of task T1 (see Figure 7.6 and Figure 7.10) and the number of question nodes in case of task T2 (see Figure 7.7 and Figure 7.12), respectively. This observation is interesting insofar that the nature of the task seems to have a higher impact on the experiment results than the characteristics of the test example itself. This is particularly true in case of task T2, whereas the case remains a bit unclear in case of task T1. This is because in case of task T1 the set of answer nodes always equates to the set of dependent objects (see Table 7.3). Hence, further studies are required to evaluate if the number of answer nodes (i.e. a characteristic of the task) or rather the number of dependent objects (i.e. a characteristic of the test example) is responsible for the increase in response time. Up to that point, the observation is considered to support the conjecture that the test examples used in this experiment are "equally complex" (in contrast to the tasks that are performed on them).

Furthermore, it can be observed that the relation between the response time and the number of answers nodes (in case of task T1) and the number of question node (in case of task T2) is different for each notation. That difference is particularly apparent in case of task T1 where an increase in response time mostly pertains to Tracematches, whereas response times of JPDDs seems to remain constant (see Figure 7.10). This gives rise to the conjecture that JPDDs scale better than Tracematches with respect to comprehension task T1. In case of task T2, the difference is not as striking, although it seems to exist (see Figure 7.12) and seems to give rise to a turning point where the one notation starts to be "better" than the other notation (see Figure 7.13).

In both cases, i.e. in case of both task T1 and T2, there is an important decline of the response times for the test examples with the highest number of answer and question nodes, respectively (see Figure 7.6, Figure 7.7, Figure 7.10 and Figure 7.12). This observation limits the previous considerations and conjectures insofar that they may only pertain up to a certain point, and that there may be another influencing variable which overrules the impact of the number of answer nodes and questions nodes, respectively, from a certain point forward.

7.6.3 Interpretations

As elucidated in the previous chapter (Chapter 6), possible reasons for the superiority of JPDDs over Tracematches with respect to an easy detection of dependent objects (i.e. task T1) could be that readers of Tracematches suffer from mentally mapping the numerous occurrences of variable names in the keywords (e.g. `target` and `returning`) of the Tracematch. In order to do so, they need to repeatedly scan the program code and determine for each object if it is mentioned in at least two (or `n`=3, 4, or 5) symbols. The increasing response times for Tracematches in case of task T1 suggest that participants have growing problems with that task. In contrast to that, JPDDs represent dependent objects using singular symbols. And thus, readers of JPDDs have to scan the specification of the join point selection only once, and check for each symbol if it has more than one relationship connected to it (in order to identify the corresponding object as a dependent object).

Apparently, the advantage of representing dependent objects as singular elements does not pay off in case of task T2 (variant `n`=2, 3, or 4) where multiple methods need to be identified which conjointly use several of these dependent objects. It seems that in this case, the sequential and redundant mention of the dependent objects in every method they are used in helps readers of the Tracematch to identify those methods. In contrast to that, readers of JPDDs need to traverse all object-to-method-relationships from the mentioned objects in order to identify the methods connected to them; afterwards, readers need to select those methods which are connected to *all* of the mentioned objects.

The experiment results suggest that the traversal of object-to-method-relationships is no problem. This is because in three of four cases, readers of JPDDs are even faster than readers of Tracematches when they need to select all methods which are connected to just *one* given object (i.e. in case of T2, variant `n`=1; see Figure 7.12). The observed increase in response time depending on the number of question nodes suggests, though, that remembering all visited methods seems to become an arduous task as soon as more (methods connected to) depending objects need to be investigated (such as in case of T2, variant `n`=2, 3, or 4).

Considering that the response times of Tracematches and JPDDs decline for test examples with the highest number of answer nodes (i.e. seven) and question nodes (i.e. four), respectively, gives rise to the conjecture that the detection of dependent objects in Tracematches (in case of task T1) as well as the identification of involved method nodes in JPDDs (in case of task T2) must get easier from a certain point forward. Any attempt to ascribe the observed decline to a characteristic of the test examples mentioned in Table 7.3 and Table 7.4 fails, though. Hence, the only explanation which sounds plausible at this point is that – in case of task T1 – participants have benefited from the fact that almost every object is a dependent object (i.e. seven out of ten), and only few (i.e. three) objects are independent. As a result, readers of the Tracematch are likely to find dependent objects in just every part of the Tracematch; i.e., they may discover several dependent objects during a single traversal of the program code. In contrast to that, i.e. in case of task T2, readers of the JPDD may have benefited from guessing when they are required to find all methods (i.e. symbols) that involve four given objects. Readers may have guessed that there cannot be many methods which share as many as four objects.

A last remark shall be made about the unexpected close-to-significant effect of the alternation on the measured response times. Considering that the assignment of the participants to groups was accomplished automatically by the apparatus, the abilities and expertise of the group members should be uniformly distributed among groups. The structure of the groups which is reported in Table 7.5 and Table 7.6 seems to confirm this expectation. Considering that the *same* group of participants, i.e. the participants starting out with JPDDs, usually produces the quicker response times makes it unlikely that these quicker response times are due to different object and method names used in the paired test examples. Hence, it seems that the alternation of notations has indeed a close-to-significant impact on the response times. No explanation can be contemplated to suggest possible reasons for this impact at this point. Thus, further exploration is needed to investigate this unexpected result.

7.7 Threats to Validity

This section identifies possible threats to the internal and external validity of the experiment. Internal validity refers to the extent to which the experiment setting actually reflects on the "cause" or "dependency" under study. External validity refers to the ability to generalize the test results to other situations.

7.7.1 Internal Validity

Threats to the internal validity of the experiment may arise from the fact that the test examples are formatted, i.e. syntax highlighting and indentation have been used to represent Tracematches, and control flow edges in JPDDs have been represented using a thicker line width. This formatting of the test examples may have a confounding impact on the measured response times. Moreover, the confounding impact may be different for each notation.

Similar confounding effects could arise from the layout of the text examples: Tracematches are laid out with the methods arranged from top to bottom (in the order they are supposed to occur in the execution trace), and with the objects being placed in the same lines as the methods they are involved in. JPDDs are laid out with the method arranged from left to right (in the order they are supposed to occur in the execution trace),

and with the objects being automatically positioned by the GraphViz tool [*Gansner & North (2000)*].

As a matter of fact, it is known that indentation has a positive effect on the comprehension of program code (cf. [*Miara et al. (1983)*]), whereas no such effect could be observed for syntax highlighting (cf. [*Hakala et al. (2006)*]). No investigations could be found concerning the impact of line width on the comprehension of diagrams. However, [*Purchase et al. (1997)*] has shown that the layout of graphs (i.e. symmetry as well as bending and crossing of lines) has a significant effect on the time needed to find shortest paths as well as vertex cuts and edge cuts of a graph. Hence, confounding effects due to formatting and layout of the test examples must be considered existent.

Nevertheless, syntax highlighting and indentation are state-of-the-art in presenting program code and most code editors (automatically) support it today. Thus, the use of syntax highlighting and indentation to represent Tracematches seems justified. It can be argued that visual notations must cope with this advantageous "standard representation" of the textual notation, and that confounding effects are therefore acceptable. Concerning the layout of the test examples, on the other hand, further studies need to be conducted in order to investigate if different layouts of the JPDDs lead to different experiment results.

Finally, confounding effects could arise from slight semantic differences between Tracematches and JPDDs concerning the specification of the methods (whose invocations are relevant to the join point specification) as well as of the type or class of the objects (which are involved in those method invocations): while Tracematches constrain the classes declaring the methods, JPDDs indicate the objects receiving the method call. And while Tracematches declare the type of the objects, JPDDs constrain the class of the objects. Apart from that, Tracematches always contain an advice specification, while JPDDs never do.

In order to prevent these differences from taking effect in the experiment, all-quantifiers ("`.*`") are used in Tracematches in order to constrain the classes which declare the relevant methods and objects are declared to be of type "Object". In JPDDs, the classes of objects are confined using all-quantifiers ("`.*`"). No constraints are specified on the target instances receiving the method calls (e.g. using the `target` keyword in Tracematches), i.e. all data dependencies are established on input and output parameters of the method calls. Finally, the advice specification of Tracematches consist of a mere invocation of an "executeadvice" method, which takes all declared variables of the Tracematch as parameters.

7.7.2 External Validity

Threats to the external validity of the experiment arise from the fact that all participants are students and that most of them are new to the notations under test. Thus, there is a risk that the inexperience of the participants and their very basic knowledge about the notations affect the test results in a substantial way. This risk is considered to be modest, though, because the experiment tasks are believed to be fairly easy and do not require much knowledge or experience about/with the notations under test in order to solve them.

Another threat to the external validity of the experiment is the complexity of the test examples. It seems reasonable to assume that the complexity represents an influencing parameter to the experiment results. Therefore, complexity has been fixed (cf. section 7.3).

In this experiment, test examples have been chosen to be rather "complex". In order to see if same or similar results are obtained for test examples of "lower" complexity, further investigations are necessary.

Finally, it needs to be emphasized that the experiment focuses on the representation of data constraints in join point selections. Hence, it remains unclear if the observed benefits of JPDDs with respect to the comprehension of join point selection constraints will also show in case of other selection constraints. Likewise, the study has concentrated on just one aspect-oriented programming language (i.e. Tracematches). Thus, there is a risk that no comprehension benefit of JPDDs will show in comparison with other aspect-oriented programming languages, or language constructs (such as the `dflow` pointcut designator [*Masuhara & Kawauchi (2003)*, *Alhadidi et al. (2009)*]). Further studies are needed to see if JPDDs facilitate the comprehension of other kinds of join point selections constraints and if they outperform other aspect-oriented programming languages, too.

7.8 Summary

This chapter has shown evidence that JPDDs are capable of facilitating the detection and comprehension of selection constraints in aspect-oriented join point selection. To do so, the chapter has selected one particular kind of selection constraint, and has formulated five questions which need to be solved in order to detect and comprehend such selection constraints. The chapter has presented an experiment design which investigates the suitability of different notations to solve those questions. Finally, it has reported on the concrete execution and on the results of such experiment which compares JPDDs with Tracematches.

The experiment has been conducted with 35 participants (divided into two groups), which are students of a computer science degree and which are mostly novice to the notations used in the experiment. Several tasks with several variants have been investigated on a basis of 28 (artificial) join point selections, which have been automatically generated. Exactly one task had to be accomplished for each join point selection. Each participant had to fulfill this task once with the join point selection being represented as a JPDD and once with the join point selection being represented as a Tracematch. The experiment evaluated the time needed to correctly solve the tasks. The experiment was evaluated using three-way ANOVA with repeated measurements on two variables.

The results of the experiment show that JPDDs facilitate the detection of dependent objects in aspect-oriented join point selections, whereas Tracematches seem to facilitate the identification of methods which conjointly use a set of (i.e. more than two) dependent objects.

7.8.1 Outlook to Next Chapter

The next chapter concludes this thesis. As such, it recapitulates the contributions of this thesis and points out interesting directions of further research based on the findings of this thesis.

Chapter 8
Conclusion

This chapter summarizes the major statements of the previous chapters of this thesis. It elucidates the achievements made and their limitations, and finally it gives an outlook to interesting future work.

8.1 Summary

This thesis addresses the problem of comprehending complex implementations of non-trivial join point selections in aspect-oriented software development. In particular, it considers join point selections which constrain several events in the execution history of a running program. The thesis points out that software developers often find themselves confronted with an extensive and detailed implementation of such join point selections, which requires careful and detailed examination in order to grasp the actual intent of the program code, i.e. of the join point selection. This burden is considered to be an important impediment to the communication among software developers. The thesis presents seven examples, each implemented in a different programming language (i.e. in AspectJ [*Kiczales et al. (2001)*], AspectC++ [*Spinczyk et al. (2002)*], AspectCOBOL [*Lämmel & Schutter (2005)*], Alpha [*Ostermann et al. (2005)*], Aquarium [*Wampler (2008)*], Perl Aspect [*Kennedy et al. (2009)*], and Tracematches [*Allan et al. (2005)*]), in order to illustrate the relevance and the universality of the problem. The thesis recognizes that readers of complex pointcut implementation suffer from the variation in language constructs, concepts, and semantics between different programming languages, as well as from the emergence of (inherent and usually non-explicit) interdependencies between the usually numerous components of a complex pointcut implementation.

The thesis asserts that existing aspect-oriented software development approaches are usually insufficient to solve this problem because they commonly provide only a limited and specific set of join point selection means. With help of these means, software developers are able to specify their join point selection means in a concise and succinct manner only in certain situations. However, in other situations, they still need to resort to complex and manual workaround implementations in order to effectuate a join point selection which they require. As a result, none of the existing approaches provides sufficient and appropriate join point selection means to express all of the motivating join point selections presented in the problem statement of this thesis.

In response to that, the thesis presents Join Point Designation Diagrams (JPDDs) as a comprehensive and visual notation to represent join point selections. As such, JPDDs are particularly capable of expressing join point selections which involve selection constraints on system events in the execution history of a program. JPDDs provide sufficient join point selection means to express all of the representative join point selections presented in the problem statement of this thesis. As a result, JPDDs are considered to be able to represent most of the aspect-oriented join point selections that can be found in aspect-

oriented literature. As a particular feature, JPDDs provide various means to reflect on different conceptual views on program execution, i.e. on a control flow-based view, a data flow-based view, as well as a state-based view. Each of these different conceptual views highlights different characteristics of the execution of a program, such as nesting of method calls, shared use of data, or oscillation between system states, etc.. By providing these different means, JPDDs react on the observation that join point selections commonly involve selection constraints that emphasize a particular conceptual view on program execution, and that any attempt to realize such joint point selections with the help of join point selection means relying on another conceptual view eventually leads to an inediquate pointcut representation.

After introducing JPDDs, the thesis discusses the benefits of JPDDs over conventional approaches by revisiting the motivating examples from the problem statement, and by comparing their implementation using aspect-oriented programming languages to their corresponding JPDDs. The thesis argues that the JPDDs are easier to comprehend because they highlight and externalize the interdependencies that must be fulfilled between all relevant events in the execution history of the program. The possibility to emphasize particular selection constraints of the join point selection depending on its underlying conceptual views on program execution are particularly helpful in that regard. Apart from that, the readers of the join point selections are freed from learning the essentials of (each of) the programming languages which are used to implement these join point selections, which means a futile waste of efforts particularly when readers only want to understand (rather than modify or evolve) the objectives of the join point selection. Of course, these benefits of JPDDs come at a cost, i.e. the necessity to learn the notational means of JPDDs themselves first. These costs are expected to pay off, however, every time that a reader needs to cope or correspond with another software developer about a join point selection which is implemented in a programming language that she/he is unfamiliar with.

Finally, the thesis investigates empirically if the conjectures made in the previous discussion may be considered to hold, actually. It does so by conducting a controlled experiment. That controlled experiment focuses on a very particular feature of JPDDs for the sake of feasibility: the experiment assesses the suitability of JPDDs to facilitate the detection and comprehension of data constraints in join point selections. To do so, it compares JPDDs with Tracematches, which are particularly designed to express data constraints in join point selections. The thesis identifies several task which need to be accomplished in order to detect and comprehend data constraints in a join point selection and asks 35 students of a computer science degree, which are mostly novice to the notations used in the experiment, to perform these tasks on a basis of 28 automatically generated join point selections. Exactly one task had to be accomplished for each join point selection. Each participant had to fulfill this task once with the join point selection being represented as a JPDD and once with the join point selection being represented as a Tracematch. The experiment measured the times needed to correctly solve the tasks in each of the notations, and evaluated their difference. The experiment results showed that JPDDs outperform Tracematches with respect to the identification of objects which are involved in data constraints, whereas Tracematches outperform JPDDs with respect to the identification of the methods which involve (several of) these objects.

8.2 Achievements and Limitations

A major achievement of this thesis is the presentation of a visual notation which may serve as a communications means for software developers and which permits them to specify, understand, and discuss about pointcut design without the need to acquire each other's programming languages or programming idioms. That presented notation consolidates many of the most influential join point selection means that are provided by current aspect-oriented programming languages. These include in particular control flow-based join point selection means (such as the `cflow` pointcut designator in AspectJ [*Kiczales et al. (2001)*]), state-based join point selection means (such as proposed by [*Douence et al. (2004)*] and implemented by JAsCo [*Vanderperren et al. (2005)*]), as well as data flow-based join point selection means (such as the ones offered by the `dflow` pointcut designator [*Masuhara & Kawauchi (2003)*, *Alhadidi et al. (2009)*] or Tracematches [*Allan et al. (2005)*]). Furthermore, the notation permits to constrain the inheritance tree (such as supported by the "+" operator in AspectJ) and the object graph (such as supported by Path Expression Pointcuts [*Al-Mansari & Hanenberg (2006)*]) – which are not considered in this thesis in greater detail since the focus of this thesis is on the representation of selection constraints relating to interdependencies between two or more system events in the execution of a program.

The thesis has presented an extensive and representative collection of join point selections mostly taken from aspect-oriented literature which all can be represented with the help of JPDDs. As a result, JPDDs are believed to be comprehensive enough to express most of the aspect-oriented join point selections that have been identified as substantive and valuable in aspect-oriented literature. However, the thesis did not conduct any investigation of the completeness or expressiveness of JPDDs in comparison to any of the existing aspect-oriented (programming or modeling) approaches. Hence, no claims can be made that JPDD be a true superset of the join point selections means provided by (some of) the other aspect-oriented approaches. And, of course, no claims can be made that JPDDs are capable of representing "just any" join point selection.

Rather than providing an all-encompassing "master" notation, the goal of this thesis is to supply software developers with a means that facilitates the comprehension of complex join point selections. The thesis has provided empirical evidence which confirms that this goal is achieved, and that JPDDs are indeed capable of facilitating the comprehension of complex join point selections at least to a certain extent. However, due to the considerable efforts needed to conduct a controlled experiment, this thesis has investigated a particular – yet well-defined – comprehension task only: the empirical evaluation concentrated on the comprehension of data constraints. Furthermore, it compared JPDDs to only one other notation (i.e. Tracematches). Consequently, it remains undecided whether JPDDs may also facilitate the comprehension of other constraints, such as constraints on control flow or constraints on states and state transitions (even though there are plausible arguments which make this seem probable; cf. Chapter 6), or that they also outperform other notations to express data constraints (such as the `dflow` pointcut designator [*Masuhara & Kawauchi (2003)*, *Alhadidi et al. (2009)*], for example).

Finally, the thesis has presented a variety of notational means which refer to different conceptual views on program execution, and which are able to emphasize different selection constraints on the execution history of a program. The thesis demonstrates why these different notational means are expected to be particularly helpful to (further) improve

the comprehensibility of selection constraints. However, the empirical investigation conducted in this thesis does not address a comparison of these different notational means. As a result, the advantages of the different notational means remain purely hypothetical at this point.

8.3 Future Work

Based on the results and observations made by this thesis, there is a couple of interesting future work which ought to be addressed next.

Above all, it would be consequent to investigate the suitability of the other notational means of JPDDs to facilitate the comprehensibility of join point selections. These are, for example, the interaction diagram-based means to represent control flow constraints and the state chart-based means to represent stateful join point selections. Again, such investigation should be accomplished empirically and in comparison to an existing aspect-oriented programming language which is capable of expressing these constraints in a likewise succinct and concise manner. The investigation could investigate if JPDDs have benefits in highlighting control flow constraints, stateful join point selections, and/or structural constraints (such as inheritance constraints or object graph constraints), too.

Furthermore, it would be interesting to assess if the different notational means to express different conceptual views on program execution actually affect the comprehensibility of a join point selection. Each of these means emphasizes different join point selection constraints. And thus, an empirical comparison of these means would show if the different emphasis of join point selection constraints actually leads to an increased comprehensibility of that join point selection constraint.

Likewise, it would be interesting to further investigate the observed correlation between the characteristics of the comprehension task and the response times which are measured to solve the task. The empirical investigation conducted in this thesis suggests that for one kind of task, response times increase with the number of entities that are given in the question, while for the other kind of task, response times increase with the number of entities that are part of the answer (in both cases, a positive correlation only shows up to a certain point; cf. Chapter 7). It would be interesting to investigate if this observed correlation can be confirmed by an empirical investigation of its own. Furthermore, it would be interesting to find out why the response times depend on different characteristics in each task.

Similarly, it would be interesting to find out the *reasons* why JPDDs are easier to comprehend than their textual counterparts. One possible approach to do so could be to identify the impacting factors on response times for each notation individually (i.e. once for JPDDs and once for the textual representation). Subsequently, a prediction model for response times could be proposed for each notation (and for each comprehension task), which could then be combined in order to estimate under which circumstances JPDDs will outperform their textual counterparts. The empirical investigation conducted in this thesis gives initial suggestions about possible impact factors for such prediction models. However, further studies are needed to find out more impacting factors. Eventually, a model can be formulated which may be used to explain the reasons for the differences in the measured response times between a JPDD and a textual counterpart with respect to a comprehension task.

8.4 Closing Words

This thesis has presented situations in which aspect-oriented programmers find themselves confronted with the need to implement complex join point selections with the help of multiple smaller components which only conjointly realize the desired join point selection. Implicit interdependencies arise between these components which substantially impede the comprehension of such join point selections, and which obstruct an efficient knowledge sharing between software developers working on the same software project and/or between software developers programming in different programming languages.

More suitable join point selection means which could shorten the implementation of such join point selections and which could facilitate their comprehension may be known in other programming languages. However, they are not at the software developer's disposal in the given programming languages, and switching programming languages may be no option for one or the other reason. In consequence, there is a need for a complementary means that may facilitate the comprehension of such pointcut implementations and that may ease the communication between software developers.

This thesis has presented a visual notation to represent join point selections, and it has shown that the notation is capable of aggregating the fragmented implementation of a join point selection into a singular and consolidated representation of the selection which explicates the important interdependencies which inhere in the join point selection. Moreover, the thesis has shown (in an empirical experiment) that the notation is even capable of facilitating the comprehension of join point selections in situations where the join point selections *can* be represented in a singular and consolidated manner even in the program code.

In conclusion, this thesis suggests the use of a complementary notation which is capable of expressing the implicit yet crucial interdependencies between multiple program fragments in order to highlight the conjoint goal of these fragments for the sake of an increased comprehensibility of the program. Furthermore, the thesis has unveiled the benefits that a *visual* notation may bring to that increased comprehensibility, even when the fragmentation of the join point selection is reduced to a reasonable and feasible degree in the program code.

Appendix A
More Motivating Examples

In addition to the examples presented in the problem statement (Chapter 3), this appendix presents further evidence that developers may be confronted with serious comprehension problems when they need to deal with complex pointcut implementations. While the examples in the problem statement have been implemented using the aspect-oriented programming languages AspectJ, AspectC++, AspectCOBOL, and Alpha, the examples presented in this appendix are implemented with Aquarium [*Wampler (2008)*], Tracematches [*Allan et al. (2005)*], and Perl Aspect [*Kennedy et al. (2009)*].

A.1 Implementing A Server Test Aspect with Aquarium

The first example is implemented with Aquarium [*Wampler (2008)*], which is an aspect-oriented toolkit for Ruby. Aquarium exploits the metaprogramming facilities of standard Ruby to weave aspects. Its major contribution is a powerful pointcut language which permits developers to specify where and when aspects shall apply in a "modular" and "user-friendly" (and AspectJ-like) way. Aquarium provides means to specify before, after, and around advice.

The following example (see Listing A.1) is adopted from [*Nishizawa et al. (2004)*], and is about a server test aspect which is supposed to verify if a request to register a new user at some authentication server actually leads to the corresponding request to add the user to the database (which is accomplished by some database server). In extension to the original scenario outlined in [*Nishizawa et al. (2004)*], the aspect presented here is also supposed to verify if user name and password are identical in both requests.

A.1.1 Explanation of the Code

In order to achieve the task, the aspect defines two advice, one wrapping around the `registerUser` request and one wrapping around the `addUser` request. The first advice maintains a control flow stack (`cflowstack`), which is evaluated by the second advice in order to determine if the `registerUser` request is still "active" (i.e. on the call stack) when the `addUser` request is being called (the length of the `cflowstack` is greater 0 in this case; see line 17).

Note that the aspect makes use of a stack data structure (rather than a Boolean flag or a counter) in order to keep track of all active `registerUser` requests. This is necessary because the aspect needs to remember the arguments (i.e. user name and password) which have been passed on with those requests. By using a stack, the second advice may peek at the top of the stack in order to find out about these arguments (see lines 18-20) so that it may compare these arguments with the arguments of the `addUser` request in its join point adaptation.

Listing A.1 Complex implementation of a join point selection in Aquarium.

```
require 'aquarium'
include  Aquarium::Aspects

cflowstack = Array.new

Aspect.new :around, :calls_to => [:registerUser], \
  :on_modules_and_descendents => [:AuthServer] \
  do |jp, obj, *args|
  cflowstack.push [args[0], args[1]]
  jp.proceed
  cflowstack.pop
end

Aspect.new :around, :calls_to => [:addUser], \
  :on_modules_and_descendents => [:DbServer] \
  do |jp, obj, *args|
  if (cflowstack.length > 0)
    exposeddata = cflowstack.last
    user1 = exposeddata[0]
    password1 = exposeddata[1]
    user2 = args[0]
    password2 = args[1]
    #join point adaptation using user1, password1, user2, and password2
  end
  jp.proceed
end
```

A.1.2 Discussion

Admittedly, the Aquarium code of this aspect looks pretty straight-forward and should be mostly self-explaining to an aspect-oriented software developer. Nevertheless, to be absolutely sure about its behavior, readers of the code need to know the meaning of all poincut predicates of Aquarium (which comes with a huge set of synonyms for each predicate). They need to learn how the predicates are deployed, how they are combined, and how they may be affiliated with the advice. Apart from that, they need to know about the meaning of the parameters of the advice code block (i.e. `jp`, `obj`, and `*args`), i.e. what they refer to. Furthermore, they need to know about the data structure which is used to store the relevant data (i.e. an `Array` used as a stack), as well as its interface.

Apart from that, readers must discover the dependencies which exist between the two advice, and they need to understand their implications. The implications are very different from what has been discussed in the examples in the problem statement (see Chapter 3). This is because the particular way in which the common data store is maintained by the first advice leads to a call stack-dependent behavior of the second advice (similar to what may be expressed with help of the `cflow` pointcut predicate in AspectJ[61]). That is, the actual join point adaptation in the second advice is performed only if the `registerUser` request addressed to the authentication server is still executing and has not terminated yet. There is no programmatic construct in the aspect code which points

[61] Note that Aquarium does not provide an equivalent to the `cflow` pointcut predicate in AspectJ.

out this fact. Consequently, readers are compelled to contemplate the behavior and purpose of the aspect by mere intellectual reflection. And they need to be very attentive not to misconceive the call stack dependency as a mere data dependency, and thus to overlook the significance of the call stack to the intents of the aspect.

Another (minor) problem may arise from the very basic context exposure capabilities of Aquarium, which requires developers to look up the definition of the intercepted methods (e.g. of method `addUser`, whose method definition is specified in another program file; it is not shown in Listing A.1) in order to find out what arguments are actually being used by the advice. Usually, readers of the program code may rely on a proper and meaningful naming of variables – which sometimes may be spared, though (such as in the first advice; see line 9).

A.2 Implementing A Decorator Test Aspect with Tracematches

Tracematches [*Allan et al. (2005)*] are an extension to the Aspect Bench Compiler (abc), which is an easy-to-extend re-implementation of the AspectJ compiler. Tracematches permit to specify a regular pattern of **symbols** (which denote program events such as method calls, for example[62]) which must occur in the execution trace of a program. Furthermore, they permit to specify free variables which are bound to particular values involved in those symbols. A join point selection will only occur if the values associated with the free variables comply to each other at each matched symbol in the execution trace (e.g. at each method call in the execution trace that matches a symbol in the regular pattern). By these means, Tracematches are particularly suited to define data dependencies (that must exist) between different events in the execution trace. Tracematches may trigger before, after, or around advice.

The deployment of Tracematches shall be exemplified with help of a decorator test aspect (see Listing A.2) which prevents nested re-invocations of decorator methods on the same decorator object [*Gamma et al. (1995)*]. A possible scenario in which this may happen is the development of a GUI application where the decorator object of a GUI widget illegally provokes a state change in the core application. In response to that, the application may perform an update of the GUI widgets, which inevitably leads to a re-invocation of the decorator method that originally provoked the state change. The decorator method may now provoke a new state change, triggering a new GUI update – and thus making the application run into an infinite loop. The goal of the aspect is to detect and intercept such infinite loops.

A.2.1 Explanation of the Code

In order to implement the decorator test aspect, the Tracematch defines three symbols, i.e. program events (see lines 5-14), which must occur in the execution trace in a particular order (as defined by the regular pattern, which is explained further below). All of the symbols relate to the invocation of a decorator method, called "print", on an "OutputDecorator" object. While the first symbol refers to the instant `before` the method invocation, the last symbol refers to the instant `after` the method termination.

[62] Other options are method executions, constructor calls/executions, field assignments/references, etc.; a symbol is defined by an AspectJ pointcut.

The middle symbol wraps `around` the selected method call (this is the event where the advice is going to be executed).

The Tracematch defines two formal parameters (see line 3) which are used to establish data constraints between the program events matching the symbols. The formal parameter `decorator` is used to determine that all matching program events must be invoked on the same instance (see `target`-constraints). The formal parameter `jp` is used to specify that program events matching the first and the last symbol need to refer to the same join point[63], i.e. to the same method call (see `let`-constraints; `let`-constraints are an(other) extension to the Aspect Bench Compiler and are not available in conventional AspectJ).

Finally, a regular expression (see line 16) defines the order in which program events matching the symbols must occur in the execution trace of a program. In this example, the regular expression specifies that a program event matching symbol `recurringPrint`[64] must occur after a program event matching symbol `beginPrint`. The absence of symbol `endPrint` in the regular expression means that no program event matching that symbol `endPrint` must occur in between the program events matching the symbols `beginPrint` and `recurringPrint`. In summary, the method invoked by the program event which matches symbol `beginPrint` must not have terminated yet when the re-invocation of that method occurs in order to select the latter as a join point.

Listing A.2 Complex implementation of a join point selection with Tracematches.

```
 1 public aspect DecoratorTest {
 2
 3   Object tracematch(OutputDecorator decorator, JoinPoint jp) {
 4
 5     sym beginPrint before :
 6       call(* *.print(..)) && target(decorator)
 7       && let(jp, thisJoinPoint);
 8
 9     sym recurringPrint around(output) :
10       call(* *.print(..)) && target(decorator);
11
12     sym endPrint after :
13       call(* *.print(..)) && target(decorator)
14       && let(jp, thisJoinPoint);
15
16   beginPrint recurringPrint
17     {
18       //join point adaptation returning some Object
19     }
20   }
21 }
```

[63] Keyword `thisJoinPoint` refers to a special variable which may be used to obtain reflective information about the current join point; it is instantiated for each join point at which it is used.

[64] The program event matching the last symbol in the regular expression – i.e. symbol `recurringPrint`, in this case – always denotes the join point where the (before, after, or around) advice is executed.

A.2.2 Discussion

Tracematches appeal by their capability to express both the chronological dependencies as well as the data dependencies between a sequence of program events (leading to a join point). The chronological dependencies are expressed by means of a regular expression consisting of symbols. The data dependencies are specified by using the same formal parameter at appropriate places in the definitions of those symbols. Hence, just as with Alpha (see section 3.1.4), readers of the aspect code are able to recognize these dependencies easily because the dependencies are made explicit in the program code.

However, before they can do so, readers need to learn the individual language constructs of Tracematches, as well as of AspectJ because the symbols of Tracematches are defined using AspectJ pointcuts (and may make use of AspectJ's `thisJoinPoint` keyword, such as in lines 7+14). Once they have done that, readers may set out to investigate the dependencies that must exist between the program events matching the different symbols.

Once the readers have assessed all dependencies, they will find out that the first and the last symbol of the Tracematch basically implement a control flow constraint, similar to the `cflow` pointcut designator of AspectJ. In principal, symbols of Tracematches may make use of the `cflow` pointcut designator. However, the `cflow` pointcut designator is unsuitable to implement the decorator test aspect because it only refers to the last (rather than all past) occurrences of a program event. In order to implement the decorator test aspect, however, it is necessary to get a hold of *all* active decorator instances (i.e. all decorator instances whose decorator methods are still executing and have not terminated yet). That is why a workaround implementation is inevitable, although it somewhat hides the presence of the control flow constraint and thus imposes an extra comprehension burden on the reader of the code.

The workaround implementation establishes a data dependency between the `thisJoinPoint` context objects of the method calls matching the first and the last symbol. Note how this leads to a significant change in the semantics of the regular expression defined in the Tracematch. Unlike standard regular expressions, which are generally incapable of expressing well-bracketed pairs of events, such as invocation and termination of a method, the regular expression in the Tracematch now distinguishes between two (otherwise equal) events which occur at different levels of the call stack. While this change in semantic of a regular expression is useful to implement the aspect at hand, it may be unexpected by the readers of the aspect and thus may lead to misconceptions about its behavior.

In summary, there are two kinds of data dependencies in the Tracematch: one which enforces the control flow constraint, and another one which realizes the actual data flow constraint. Readers of the code may fail to recognize the distinction between these two kinds of data constraints and thus may misinterpret the purpose and the goal of the aspect. Moreover, the usage of a regular expression to express the order in which the program events need to occur disguises the immediate nesting of the method calls which results from the specified control flow constraints. As a result, it is not easy for maintainers to read and understand the join point selection, and to assess what join points are going to be selected in the end.

A.3 Implementing A Caching Aspect with Perl Aspect

Another example shall be presented using Perl Aspect [*Kennedy et al. (2009)*]. The Perl Aspect module uses the highly dynamic capabilities of (standard) Perl to wrap arbitrary methods. The goal of the package is to provide a consistent interface which facilitates the definition of aspects and permits to cleanly separate their code from the affected classes. The Perl Aspect module permits to identify the methods which shall be wrapped by providing a particular method name, a regular expression, or a reference to a function which decides whether or not a given method shall be wrapped. The wrapped methods may be augmented with before and after advice (the before advice may also be used to implement around advice behavior).

The following Perl code (see Listing A.3) implements a caching aspect which augments a complex graph utility function (i.e. the computation of strongly connected components) with caching functionality. The goal of the aspect is to spare the re-execution of the complex computation and to re-use the previously computed result, instead.

A.3.1 Explanation of the Code

The pointcut of the aspect is defined by a regular expression and selects all calls to methods in the Perl Graph module [*Hietaniemi (2009)*] whose names end with "strongly_connected_component_by_vertex" (see line 3).

The aspect defines a Perl hash to cache the relevant data (`%chache`; see line 5). That cache will store references to other (anonymous) hashes, which will than point to the cached data. This way, the return value of the complex computation can be stored for a *pair* of input values, i.e. the graph and the vertex, in this case. See in line 12, for example, how the advice adds a new entry to the `%chache` hash with the graph object as a key and an (empty and anonymous) hash as the value (the graph object is retrieved from the join point context object by invoking `$context->self`, which returns the currently executing object)[65]. Subsequently, in line 13, the (anonymous) hash is augmented with a new entry which maps the vertex object (which is the second[66] argument of the intercepted method, and which may be retrieved by invoking `$context->params->[1]` on the join point context object) to the return value of the originally intercepted method (which has been previously stored in the local variable `$return_value` in line 10).

The aspect implements two advice which both apply to the same pointcut. The first advice is responsible for filling the cache (see lines 7-16). The second advice is responsible for retrieving data from the cache (see lines 18-28). The two advice could have been implemented as a single advice, just as well. However, the programmer has decided not to do so because he/she wanted to emphasize that the aspect is about a recurring invocation of the same method (rather than about alternative processings of a single invocation). It is important to note how the second advice instructs the join point context object (`$context`) to return the cached value to the calling context (see line 23) – and that, by doing so, the advice also prevents the originally intercepted method from begin executed[67].

---

[65] Note that this execution step is skipped if the key already exists in the `%chache` hash and if it is initialized with another value than 0 or false (see `unless` constraint at end of line 12).

[66] In object-oriented Perl, the first parameter of a method is always a `self` reference to the object.

[67] Analogously, the `else` block of the `if`-statement could be spared in this `before` advice which would mean that the Perl interpreter continues normally with the execution of the intercepted method.

Listing A.3 Complex implementation of a join point selection with Perl Aspect.

```
1 use Aspect;
2 {
3   my $pointcut = call qr/Graph::.*strongly_connected_component_by_vertex$/;
4
5   my %chache = ();
6
7   before {
8     my $context = shift;
9
10    my $return_value = $context->run_original;
11
12    $chache{$context->self} = {} unless $chache{$context->self};
13    $chache{$context->self}->{$context->params->[1]} = $return_value;
14
15    $context->return_value($return_value);
16  } $pointcut;
17
18  before {
19    my $context = shift;
20
21    if ($chache{$context->self} && $chache{$context->self}->{$context->params->[1]}) {
22      #join point adaptation:
23      $context->return_value($chache{$context->self}->{$context->params->[1]});
24    } else {
25      my $return_value = $context->run_original;
26      $context->return_value($return_value);
27    }
28  } $pointcut;
29 }
```

A.3.2 Discussion

In order to understand the Perl code shown above, readers must get used to the Perl way of doing things. In this case this means in particular to understand the utilization and handling of hashes (which is one of the two "ultimate" data structures to deal with data collections in Perl). Moreover, the aspects makes use of hashes (`%chache`; see line 3) and *references* to (anonymous) hashes (see line 12). Readers need to learn the difference before they can completely understand the implementation.

Apart from that, readers need to apprehend the semantics of the `before` advice and the join point context object in Perl Aspect. Basically, both advice in the Perl code shown above implement `around` behavior (and would have been implemented as `around` advice in other aspect-oriented programming languages, such as AspectJ). The confusion of the reader would be perfect if the `else` block of the `if`-statement of the second advice would be missing (which is possible, and would be the "Perl way", in fact) because it would make the second advice behave like a `before` advice in some cases, and like an `around` advice in other cases. Reason to this "hybrid" behavior is the special semantic of the `return_value` operation of the join point context object. Without acquiring the semantic of this method and the other methods of the context object, readers are likely to misinterpret the functioning of the aspect. An example of another method which might confuse the readers is the `self` method of the context object, which returns the owning,

and currently executing, object of the intercepted method (rather than the context object, itself). Finally, readers may misinterpret the order in which the advice affect the intercepted method. Advice in Perl Aspect are arranged in wrapping order rather than in execution order (such as in other programming languages, such as AspectJ). This leads to an "inverted" execution order of the two `before` advice (i.e. bottom-up rather than top-down).

It is not before the reader has acquired all these peculiarities of Perl and of the Perl Aspect module that he/she may start to investigate how the behavior of the one advice influences the behavior of the other advice. As in the previously presented examples, the influence results from the shared access to a common data store, and the developer needs to find out which advice writes (what elements) to that data store and which advice reads (these elements from) the data store (again). It is not before then that developers are able to identify the order in which each of the advice executes (at recurring invocations of an intercepted method rather than at a single invocation of it!). And they will recognize under which circumstances the one advice will prevent the execution of the other advice. And finally, they are able to correctly estimate the behavior and the purpose of the entire aspect.

A.4 Summary

This appendix has presented additional examples which illustrate the problems that this thesis is focusing on. By doing so, the appendix has complemented the problem statement (see Chapter 3) by (a) taking into account additional aspect-oriented programming languages (i.e. Aquarium, Tracematches, and Perl Aspect), and (b) discussing an additional issue which may give rise to complex implementations of join point selections (i.e. the specification of call stack dependent join point selections).

Appendix B

Prototypical Tool Support

This appendix presents prototypical tool support that has been provided for JPDDs. That tool support includes a UML profile which can be loaded into conventional UML modeling tools in order to create JPDDs with help of such tools, an Eclipse plug-in which is capable of visualizing JPDDs appropriately (i.e. using the JPDD-specific symbols), as well as a code generator which is capable of generating aspect-oriented program code from JPDDs. The presentation in this section is meant to be rough. More details about the tools can be found in [*Bartelheimer (2006)*, *Avramova (2008)*]. Links to the homepages of the mentioned frameworks and tools are listed at the end of this thesis. Note that some of the features of JPDDs described in this sections are not considered in this thesis.

B.1 UML Profile

This section gives an overview to the UML profile and illustrates how the profile can be used with a conventional UML modeling tool.

B.1.1 Profile Overview

JPDDs are based on the UML. Hence, it is quite straightforward to define a UML profile for JPDDs. Table B.1 gives an overview to that profile. The table shows what elements of the UML are extended (and how) in order to represent the specific notational means of JPDDs.

First of all, an «isJPDD» stereotype is defined which extends UML Package in order to denote that all contents of that respective package make up a JPDD.

Furthermore, an «identified» stereotype is defined which extends UML Element (Element is the topmost metaclass in the UML metamodel). That stereotype may be assigned to any element that is provided with an identifier. The name of that identifier may then be stored in a special tagged value, named "identifier".

Another important stereotype is «indirect». That stereotype is used to denote that a particular relationship (i.e. a UML Generalization, a UML Association, a UML Message, or a UML Transition) is meant to denote a path, rather than a direct relationship between two elements. The stereotype provides two tagged values, "lowerBound" and "upperBound", which store the minimum and maximum number of relationships (i.e. "hops") that need to be traversed on the path.

The next three stereotypes refer to the different combination relationships that may exist between two JPDDs (cf. section 5.4 in Chapter 5) The stereotype «union» denotes that the selection result of one JPDD is unified with the selection result of another JPDD. The «confinement» stereotype is used to express that the selection result of one JPDD is

confined by the selection result of another JPDD. And the «exclusion» stereotype indicates that the selection result of one JPDD is diminished by the selection result of another JPDD. All three stereotypes come with a special tagged value, named "mapping", which specifies in detail how the union, the confinement, or the exclusion has to be accomplished (i.e. which identified element of one JPDD has to be mapped to which identified element of the other JPDD). If that tagged value is left undefined, the mapping is based on identifier names (for further rules concerning the mapping definition, please refer to [*Stein et al. (2005)*].)

Table B.1 JPDD profile specification.

Stereotype	**Tagged Values**	**UML Metaclass (Base)**
isJPDD		Package
identified	identifier	Element
indirect	lowerBound upperBound	Generalization Association Message Transition
union	mapping	Dependency
exclusion	mapping	Dependency
confinement	mapping	Dependency
identifier	identifier	TemplateParameter

The last stereotype «identifier» exists for mere technical reasons. The stereotype is somewhat the counterpart of the stereotype «identified». It extends UML TemplateParameters (which are used in JPDDs to render the parameters in the output parameter box in the lower right corner of a JPDD). It provides a tagged value "identifier" which stores the name of the identifier of an element that should be exposed by the JPDD. Usually, i.e. in standard UML, the relationship between a template parameter and the element representing that parameter inside a template is rendered by means of a metarelationship between the UML TemplateParameter metaclass and the UML ParameterableElement metaclass. Unfortunately, though, not every UML Element is a UML ParameterableElement. Since in JPDDs, however, an identifier can be given to every UML2 Element, a workaround had to be found to establish a relationship between the identifier in the parameter box (i.e. the UML2 TemplateParameter) and the identified element (i.e. some UML2 Element, which is stereotyped with «identified»). A common example in JPDDs which would frequently cause a problem when being exposed as a parameter is a UML Message (such as the exposure of message "<?jp>" in Figure B.7).

B.1.2 Deploying the Profile in Modeling Tools

The profile outlined in Table B.1 has been implemented with help of the Eclipse UML2 framework. That framework is supported by an increasing number of conventional UML modeling tools. Examples are Rational Rose, Magic Draw, and others. Once loaded

into one of these modeling tools, the profile can be applied to models created with these tools, and its stereotypes can be applied to the respective elements[68].

Figure B.1 shows the profile loaded into the **Rational Software Modeler** (see "Project Explorer" in left column). On the right, a sample **JPDD** is shown, modeled in the **Rational Software Modeler** with help of that profile. A freeform diagram is used to combine a sequence diagram with an (excerpt of a) class diagram in one drawing. A package symbol has been used to visualize the border surrounding both the sequence diagram and the (excerpt of a) class diagram, and to accommodate the output (template) parameters of the **JPDD**. The «indirect» stereotype has been attached to the generalization relationship in the class diagram, as well as to the middle messages in the sequence diagram. The «identified» stereotype has been applied to the left message, which represents the join point to select, as well as to the type of the second lifeline, which is used to "connect" this type to the subclass shown in the class diagram. Furthermore, the «identified» stereotype has been applied to the **UML Property** which is represented by the second **UML Lifeline** in order to expose it as part of the output (template) parameters of the **JPDD**. Unfortunately, the Rational Software Modeler does not visualize this stereotype application in the sequence diagram. Other problems with the visualization of stereotype applications are discussed in the following section.

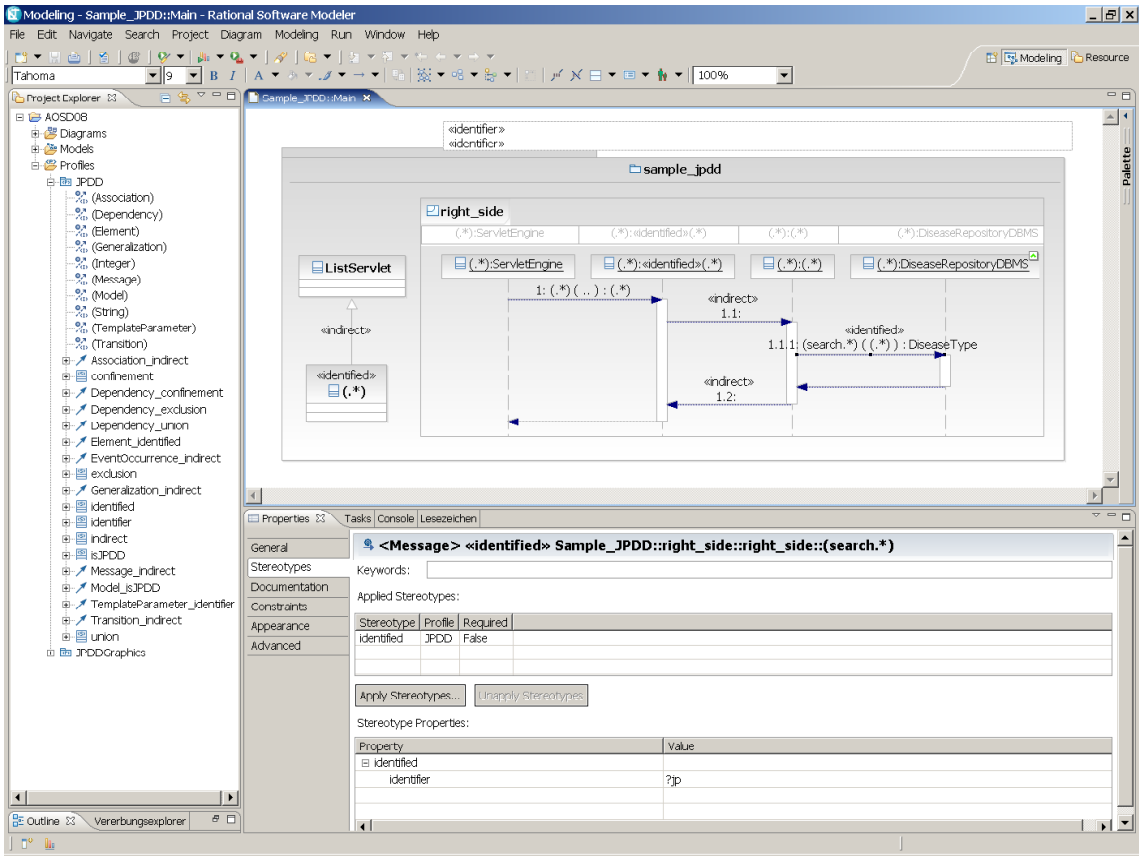


Figure B.1 Deployment of the UML2 profile in the Rational Software Modeler.

[68] Depending on the version of the UML2 framework that the respective modeling tool is resorting to, some profile conversion is required; this is accomplished as easy as opening the profile with the UML2 editor that comes with the respective UML2 framework, and saving the profile as a new file.

B.2 Eclipse-Plugin

One problem of conventional modeling tools is that they do not render JPDDs appropriately. For example, instead of visualizing indirect relationships (such as indirect generalization relationships, or indirect messages, etc.) using the JPDD-specific indirection symbol, they use a conventional UML symbol (e.g. a generalization relationship, or a message, etc.) and adorn it with a string, i.e. "«indirect»" in this case, in order to denote the corresponding stereotype application. Furthermore, template parameters are shown in the upper right corner rather than in the lower right corner of a JPDD, and their reference to a particular element within the JPDD is hidden away in the tagged value "identifier" of the «identifier» stereotype application to the respective template parameter. Likewise, the values of other stereotype tags, such as the value of tag "identifier" of stereotype «identified», are rendered in the diagram's property sheets rather than in the diagram itself. Finally, it may be the case that not all stereotype applications are shown – such as the «identified» stereotype application to UML Properties of UML Lifelines, as it is the case in the Rational Software Modeler (see section B.1.2).

Hence, a graphical modeling tool is needed that visualizes JPDDs as they are supposed to look like. Looking for a way to do so, the decision was made for the Eclipse Graphical Editing Framework (GEF) – not so much because this was the easiest way, but rather because direct integration into the Eclipse IDE promises easy utilization of the tool by the aspect-oriented software developer (considering that most aspect-oriented programming language provide dedicated development tools for the Eclipse IDE). Furthermore, GEF was designed to interplay with the Eclipse Modeling Framework (EMF), which UML2 is an instance of. Hence, it was possible to use the UML2 framework to create, manipulate, and store JPDDs. Using the UML2 framework was considered appealing since it promised interoperability between the JPDD modeling tool and conventional UML modeling tools, and it permitted to re-use the UML2 profile described in the previous section.

B.2.1 Five Plug-Ins, One Feature

The Modeler for Join Point Designation Diagrams (M4JPDD) is an Eclipse feature consisting of five Eclipse plug-ins (see Figure B.2), each one taking responsibility for a different task. In the following, the plug-ins are introduced in closer detail.

de.unidue.icb.dawis.jpdd.gef is the main plug-in of the M4JPDD. It implements all necessary Model-View-Controller (MVC) [*Buschmann et al. (1996)*] classes of the modeling tool following the guidelines of GEF (i.e. by realizing the interfaces of the org.eclipse.gef framework; see Figure B.2). Furthermore, it realizes the actual integration of the M4JPDD tool into the Eclipse platform: it provides an editor for JPDDs and associates the file extension ".jpdd" to this editor; it adds a new "newWizard" for the initial creation of a JPDD, which is invoked via Eclipse's "file" menu and the "new" item; finally, it adds a new perspective "JPDD Modeling" that combines certain views that are of interest when working with JPDDs. To do so, the plug-in extends the corresponding extension points of the org.eclipse.ui plug-in (see Figure B.2).

The classes implementing the visualization (i.e. the View) of the data structure (i.e. the Model) of JPDDs have been extracted into their own plug-in, named *de.unidue.icb.dawis.jpdd.draw2d*, in order to enable future re-use of the classes in other

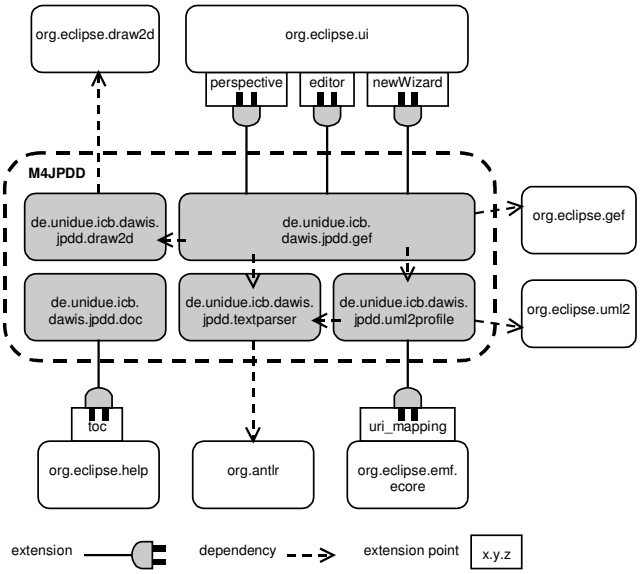


Figure B.2 General architecture of the M4JPDD Eclipse plug-ins (cf. [*Bartelheimer (2006)*]).

contexts. The visualizations have been implemented with help of the Eclipse Draw2D Framework (see org.eclipse.draw2d in Figure B.2).

The *de.unidue.icb.dawis.jpdd.uml2profile* plug-in is mainly responsible for processing the UML2 model representation of a JPDD. The plug-in contains several helper classes that provide important services to the de.unidue.icb.dawis.jpdd.gef plug-in. For example, the plug-in provides facilities to initialize, load, and save JPDD models and to provide a single point of access to the root model element in the JPDD file (to do so, the plug-in extends the "uri_mapping" extension point of org.eclipse.emf.ecore; see Figure B.2). Another group of convenience methods retrieves string representations for model elements, by traversing the underlying JPDD model structure – e.g. to visualize an operation signature (with its visibility modifiers, its parameters, and their types, etc.) inside a class symbol. To do so, the plug-in makes use of the org.eclipse.uml2 framework (see Figure B.2).

The plug-in *de.unidue.icb.dawis.jpdd.textparser* realizes a unique and highly convenient gimmick of the M4JPDD tool. Other than conventional UML modeling tools, which provide extensive property sheets with numerous textboxes and checkboxes for creating or modifying model elements (such as attributes, operations, messages, etc.), the M4JPDD provides a simpler way to specify these properties – purely textual – with the help of the keyboard. The user input is analyzed by a parser and all necessary UML model elements are created/modified according to the parsed input. The parser is realized with the help of ANother Tool for Language Recognition (ANTLR), a tool for parser generation (which made available in Eclipse via the org.antlr plug-in; see Figure B.2). The grammars implemented with that tool comply in most parts with the grammars specified in the UML specification (some adoptions had to be made). The actual creation/modification of the UML model elements is performed by the de.unidue.icb.dawis.jpdd.uml2profile plug-in. The interaction between the textparser and the uml2profile plug-in is realized using a simple data structure – consisting of mere container objects for the parsed values – which is created by the parser and forwarded to the convenience methods of the uml2profile plug-in.

Figure B.3 illustrates the feasibility of this device. The figure shows the M4JPDD Class Editor, which can be used to define several features (i.e. attributes and operations) of classes all at once. It is a very convenient way to define classes without the need to fill out bulky property sheets and to choose entries from extensive pop-up menus.

The *de.unidue.icb.dawis.jpdd.doc* plug-in does not add any new functionality to the M4JPDD tool. It just contains help files. These files are specified using the HyperText Markup Language (HTML) and are hooked into the central help system of Eclipse.

Figure B.3 M4JPDD class editor.

B.2.2 Deploying the JPDD Modeling Tool

Figure B.4 illustrates how the M4JPDD tool looks like and how it may be used to draw JPDDs. The M4JPDD tool provides means to specify structural JPDDs, interaction diagram-based JPDDs, activity diagram-based JPDDs, and state machine diagram-based JPDDs (see corresponding drawers of the tool palette shown in Figure B.4, next to the JPDD). It also allows to combine them with help of combination relationships (see "combination" entry in the "General" drawer of the tool palette shown in Figure B.4). As such, the M4JPDD tools is capable of rendering JPDDs in their proper way:

It uses a graphical indirection symbol to render indirect relationships (such as the indirect generalization in the structural specifications on the left, or the indirect message in the behavioral specifications on the right of the JPDD). It visualizes the identifier names of elements (i.e. the values of the "identifier" tag of «identified» elements) right in front of the respective names of the elements. Last but not least, the M4JPDD appropriately places the output (template) parameter box at the lower right corner of the JPDD, and displays all referenced identifier names (i.e. the values of the "identifier" tag of each «identifier» template parameter).

Before deploying the M4JPDD tool in Eclipse, it needs to be installed – either by downloading the .jar-files of the plug-ins mentioned in the previous section and by copying them into the plug-in directory of the respective Eclipse installation, or by using the Eclipse Update Manager (i.e. by selecting items "Software Updates" – "Find and Install..." from Eclipse's "Help" menu) and downloading the M4JPDD feature from the remote update site at http://dawis2.icb.uni-due.de/jpddtools/updatesite.

Once the M4JPDD features is installed in Eclipse, a new JPDD can be created by selecting the "New" – "Other..." item from Eclipse's "File" menu (or by pressing Ctrl+N), and by choosing "Join Point Designation Diagram" from the "JPDD" folder. The JPDD editor will now open (see Figure B.4), showing an empty drawing pane.

Then, a JPDD needs to be created by choosing "jpdd" from the "General" drawer of the tool palette on the left window border of the editor, and dragging the mouse pointer to

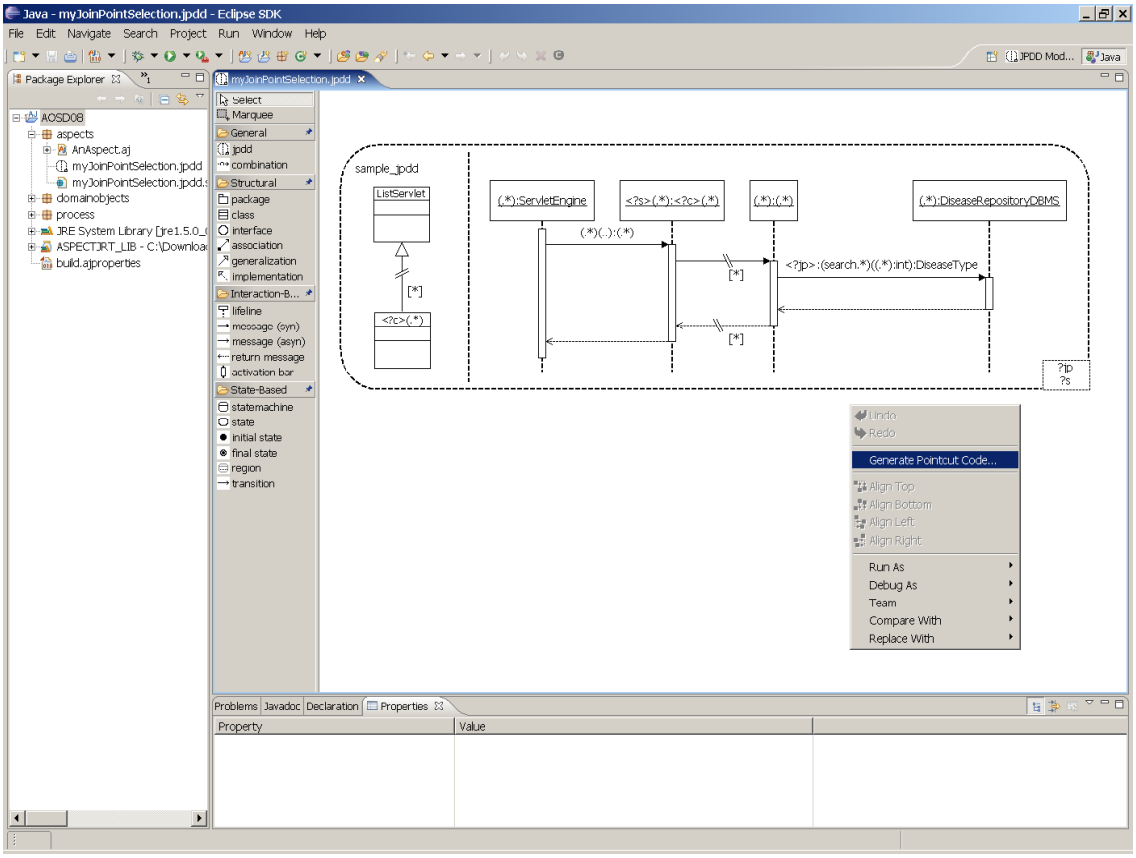


Figure B.4 The Modeler for Join Point Designation Diagrams (M4JPDD) in Eclipse.

the desired location on the empty drawing pane. Once a JPDD has been created this way, the join point selection and its join point selection criteria can be specified using the respective elements of the palette.

Using the M4JPDD tool, JPDDs can be printed out using Eclipse's standard printing facilities. More importantly, though, JPDDs can be exported to image files with help of another Eclipse plug-in, called ImageExport. With that third party plug-in, JPDDs can be exported to BMP, JPEG, PDF, and SVG file formats via the "Export..." item of Eclipse's "File" menu.

Furthermore, JPDDs can be transformed to aspect-oriented program code, which will be discussed in closer detail in the following section.

B.3 Code Generator

Having a tool to draw and visualize JPDDs appropriately and being able to print them and convert them to image files may be useful for communication and documentation purposes. In order to exploit JPDDs in actual software production, however, code generation facilities are desirable. Provided with appropriate code generators for their target programming languages, developers can benefit from the expressive power of JPDDs – irrespectively of the possible incapabilities of their target language since the code generator will build all appropriate workarounds for them. In this section, we take a closer look at the technical side of pointcut generation from JPDDs and exemplify the requirements, which

UML2 model representations of JPDDs must fulfill so that they can be successfully transformed by the code generator.

B.3.1 Two-Step Process for Code Generation

The code generation is accomplished in two steps (cf. Figure B.5). In a first step, the information contained in the UML2 representation of JPDDs is condensed to a simpler model which consists of fewer classes, and which is targeted towards the needs of code generation. Then, in a second step, that simpler model representation of JPDDs is evaluated by the actual code generator, and is transformed into program code. The following explanations concentrate on the first step as this is considered to be of greater importance to the general audience (as compared to the elucidation of the code generation steps to a concrete aspect-oriented programming language).

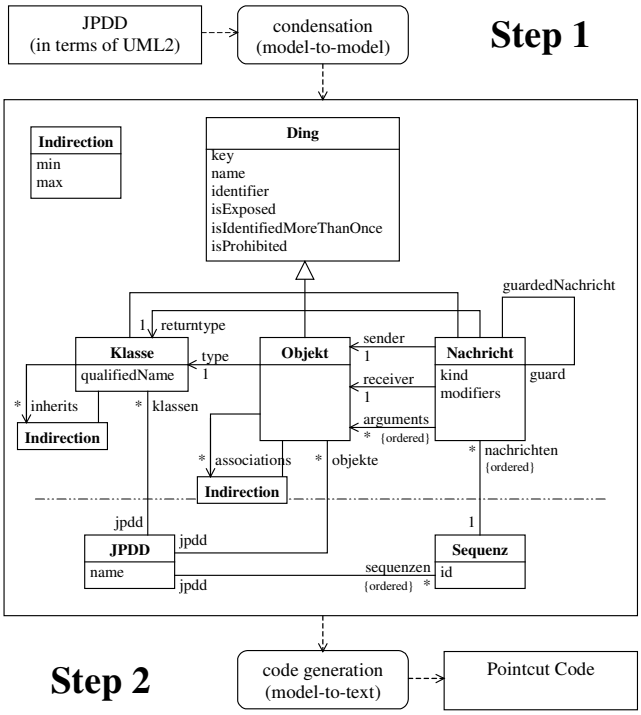


Figure B.5 Code generation process and used data model.

While creating the simpler data model in the first step, unification of identified element is performed. This is accomplished by providing each element in the simpler data model with a unique key (which, in case of identified elements, involves the identifier name). That key is used to detect whether or not two instances of an element are considered identical. If – during model condensation – an element is detected which has the same key as an element which has been previously processed (and thus is already existent in the respective data structure), the properties of both elements are unified. Unification of objects triggers unification of their types (if these types have been provided with different identifiers, this is considered an error and an exception is thrown). No unification is performed for message elements. Detecting two messages with the same identifier is considered an error and an exception is thrown (the same message cannot occur twice in one execution trace; however, the same operation can certainly be invoked by two different messages!).

Figure B.5 outlines the simpler data model which is instantiated during the first step of the code generation process. The top most element of that data model is a "Ding" (German for "thing"[69]). It provides field members (and corresponding setter and getter methods) to store the respective element's key, its name (usually a regular expression) and its identifier (if applicable). Furthermore, it stores whether or not the element is exposed in the JPDD's output parameter box (via its identifier, if applicable), whether it is identified more than once (this information may be of importance when building workarounds during the code generation step), and whether it is prohibited (i.e. constrained with a "`{not}`" constrain[70] in the JPDD).

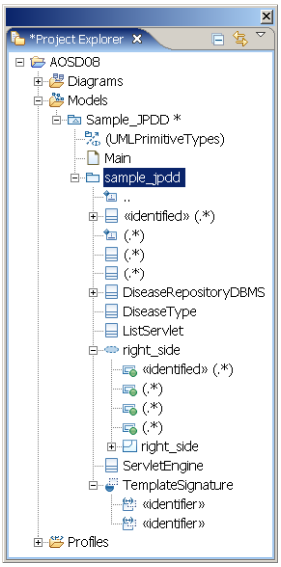


Figure B.6 Required UML2 model structure for automatic code generation.

Three subclasses inherit those members from the superclass "Ding": "Nachricht", "Objekt", and "Klasse" (which is German for "message", "object", and "class", respectively). The field members and associations of those classes should be quite self-explaining. Worth explaining are the presence of the "guard : guardedNachricht" association, as well as the purpose of the "Indirection" class, though. The former plays an important role in synchronizing different execution traces of a complex communication protocol, i.e. where one system event of one trace needs to occur before another system event of another trace, which may be subject of a complex join point selection specified by a JPDD (see Figure B.7 for an example). The latter is needed since associations between objects, and inheritance relationships between classes, may be indirect. Therefore, class "Indirection" is used as an intermediate in order to store the minimum and maximum number of association edges that may be traversed on a path from one object to the other. In case of a direct relationship, both fields "min" and "max" are set to 1.

All objects and classes which have been processed during the model condensation step are related to one instance of class "JPDD", which is the general entry point to the simpler model representation of JPDDs. It holds the (unified) sets of objects and classes, as well as

[69] German words haven been used as class names in order not to conflict with standard Java classes such as Class, Object, etc.

[70] Note that the OCL constraints `{not}` and `{or}` are not taken into account in this thesis; they have been used within JPDDs in former publications; their semantics can be realized with help of combination rules, though, as described in section 5.4.

a list of sequences, i.e. instances of class "Sequenz" (German for "sequence"), each representing one execution trace of messages as it is specified in the JPDD.

B.3.2 Deploying the Code Generator

Deploying the code generator from within the M4JPDD tool is as easy as clicking on the background of the drawing pane of the JPDD editor, and choosing "Generate Pointcut Code..." from the pop-up menu.

However, the code generator can also be executed from the command line prompt, e.g. to generate code from UML2 models created with other tools than the M4JPDD. In order to make this be successful, though, the UML2 model structure of the JPDD being fed to the code generator must satisfy certain requirements. These requirements mostly pertain to the way in which JPDDs are encapsulated in the UML2 model.

As illustrated in Figure B.6 (which is the UML2 model representation of the JPDD shown in Figure B.1), the code generator requires that the elements of a JPDD are encapsulated by a UML2 Package (stereotyped with «isJPDD»), and that these packages are immediate descendants of the top UML2 Model element. Depending on the kind of JPDD, the UML2 Package may either contain one UML2 Activity when rendering an activity-based JPDD, xor one UML2 Interaction (contained in one UML2 Collaboration) when rendering an interaction-based JPDD, xor one UML2 StateMachine (contained in one UML2 Class) when rendering a state-based JPDD, xor none of those when rendering a structural JPDD (see [*Avramova (2008)*] for further explanations on these different model structures). Within these elements, all descendant elements are to be arranged according to the UML specification. Structural information that needs to be stored together with activity-based, interaction-based, or state-based JPDDs is supposed to be located in the encapsulating parent UML2 Package.

B.4 Limitations

The prototypical tools presented in this appendix are meant to provide software developers with a rudimentary and methodical way to define and use JPDDs. The motivation is to demonstrate how JPDDs can be used seamlessly in daily software development. And therefore, the JPDD profile presented in this chapter allows for seamless use of JPDDs with conventional UML modeling tools, while the M4JPDD tool allows for seamless use of JPDDs in Eclipse (which is the target IDE of many aspect-oriented development tools, such as AJDT, JAsCoDT, etc.).

Moreover, the profile implementation with the Eclipse UML2 framework ensures (limited) inter-operability between different modeling tools (including the M4JPDD tool): the XMI files created with the Eclipse UML2 framework are supported by an increasing number of UML modeling tools. And thus, JPDD files generated with any of such modeling tools may be exchanged between the tools, and may be used to generate pointcut code from JPDDs with the code generator as described above. It is important to note, though, that exchangeability does not include the graphical information contained in JPDD files since most UML modeling tools implement their own proprietary solution for storing location and size, etc., of diagram elements (the M4JPDD tool makes use of a special stereotype to do so).

Being mere prototypes, the tools described here are not thoroughly tested and do not implement all of the features which one would desire. Important limitations apply in particular to the automatic code generation tool. These limitations are summarized next.

B.4.1 Efficiency of Generated Code

The code generator does not strive for generating highly efficient program code. Instead, the goal of the tool is to generate code that complies to the principles of JPDD translation (cf. [*Hanenberg et al. (2007)*]), which demand that the generated code helps developers to understand the general selection semantics of JPDDs. It is left to the aspect-oriented compiler to remove possible superfluous join point selection constructs and to optimize the join point selection for a given target system.

B.4.2 Completeness of Generated Code

Translation of JPDDs into aspect-oriented program code is not comprehensive. This means that the generated aspects may not reflect on *all* join point selection criteria specified in a JPDD. Selection constraints that are not translated into code, for example, are structural constraints which oblige the presence of particular operations and attributes in class definitions. Furthermore, inheritance hierarchies are considered only up to one (possibly indirect) level upwards. The code generator indicates any incapability to translate such join point selection constraints with the help of appropriate comments in the program code. Extending the code generator so that it will consider *all* join point selection constraints provided by JPDDs is subject of interesting future work.

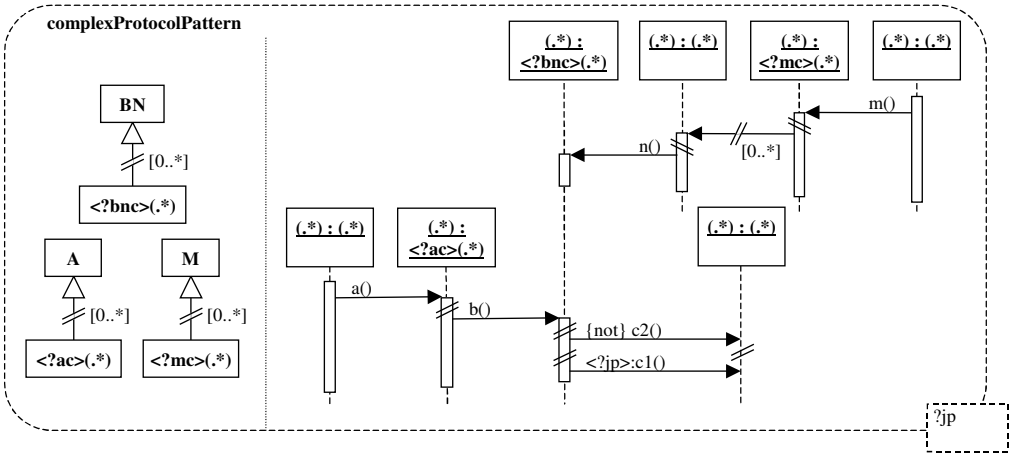


Figure B.7 JPDD defining a complex protocol.

B.4.3 Total Selection vs. Local Selection

When specifying join point selections on complex protocols (such as the one presented in Figure B.7)[71], semantic mismatches between the JPDD and the generated code might result from the fact that only the first occurrence of a given execution trace pattern is considered. This becomes problematic whenever one trace pattern reposes on the data (e.g. object references) previously collected by another trace pattern. In that case, the later trace

[71] Note that the JPDD shown in Figure B.7 makes use of features which are not taken into account in this thesis.

pattern may fail to select a join point since it neglects all data that would have been collected in later matches of the previous trace pattern. In order to solve this issue with AspectJ, a data structure would be necessary that collects the state information of all matches of the trace pattern together with the data being involved in that match. An alternative approach would be to use (a combination of) Tracematches to implement the correct behavior. Extending the code generator so that it will implement this behavior is subject of interesting future work.

B.4.4 Reverse Engineering Facilities

The prototypical tools also provide limited reverse engineering facilities. However, these facilities only work for simple join point selections consisting of a single pointcut. No facilities are provided to generate JPDDs from a collection of pointcuts and advice. Neither, it is planned to implement such feature in near future. It is believed that reverse engineering of aspect-oriented pointcut code to JPDDs is possible only in the simplest cases. Furthermore, it is believed that in these cases, JPDDs do not add very much to the comprehensibility of join point selections: the benefits of JPDDs are the greatest when the aspect-oriented programming language in use does not support a requested join point selection means, i.e. whenever workarounds must be implemented that consist of a group of poincuts and advice. In such cases, JPDDs can help to exhibit the original intent of the join point selection. However, the number of valid workaround implementations is high, and can vary significantly from each other, so that no general reverse engineering algorithm can be found to create a JPDD from the (many) pointcuts and advice of the workaround implementation.

B.5 Summary

This appendix has presented prototypical tool support which may be used to create, save, and load JPDDs as well as to generate aspect-oriented program code from them. The modeling tool supports all of the notational means presented in Chapter 5. The code generator is capable of generating rudimental aspect-oriented program code from interaction diagram-based JPDDs, only. The tools are ready for experimental use in the software development process. Further implementation work is required for a productive deployment, though. The tools can be downloaded from http://www.dawis.wiwi.uni-due.de/en/research/foci/aosd/jpdds/.

Appendix C

More Facts and Figures

This appendix complements the evaluation chapter (Chapter 7) with additional figures and data tables. These are in particular a visualization of the measured response times, the estimated marginal means of the measured response times for each treatment of the independent variables (i.e. "notation", "alternation", and "test example") together with their confidence intervals, as well as the estimated marginal means of the differences of the measured response times (divided by alternation) together with their confidence intervals.

Figure C.1, Figure C.2, Figure C.4 and Figure C.5 give a visual impression of the response times measured for the different tasks using the different notations. Figure C.3 and Figure C.6 gives a visual impression of the differences between those response times measured for the different notations. Each figure discriminates between data points obtained from participants beginning with JPDDs ("JPDDfirst") and data points obtained from participants starting with Tracematches ("TMfirst"). All figures mark quantiles and means for each data column (see gray lines next to data points). All data is measured in milliseconds (ms).

When investigating Figure C.1, Figure C.2, and Figure C.3, mind the different scales: the slowest response time of task T1 when using Tracematches is approximately three times slower than the slowest response time of task T1 when using JPDDs, whereas the slowest response time of task T2 using Tracematches is roughly five times quicker than the slowest response time of task T2 when using JPDDs.

Table C.1, Table C.2, Table C.3, Table C.6, Table C.7, and Table C.8 show the estimated marginal means of the response times of both tasks T1 and T2 for each treatment of the independent variables together with their confidence intervals. Likewise, Table C.5 and Table C.10 give the estimated marginal means of the *differences* of the response times of both tasks T1 and T2 and their confidence intervals (divided by alternation).

Figure C.7 shows a bigger and thus more readable version of the test example shown in Figure 7.2 (on page 114).

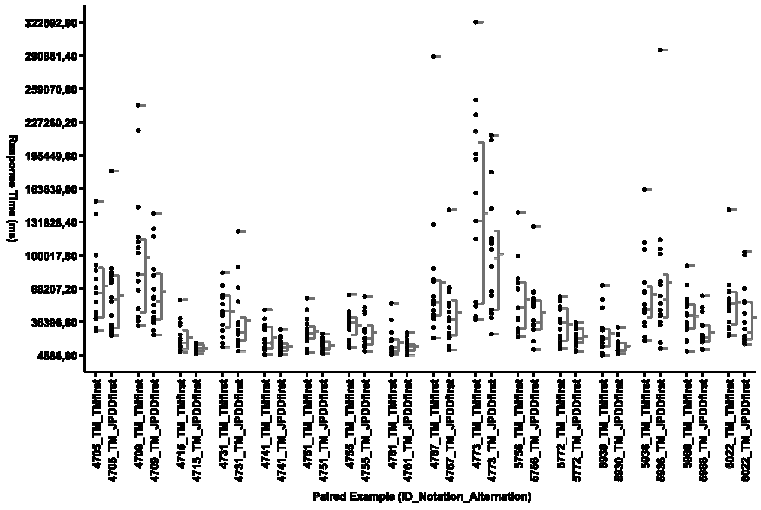


Figure C.1 Response times of task T1 using Tracematches.

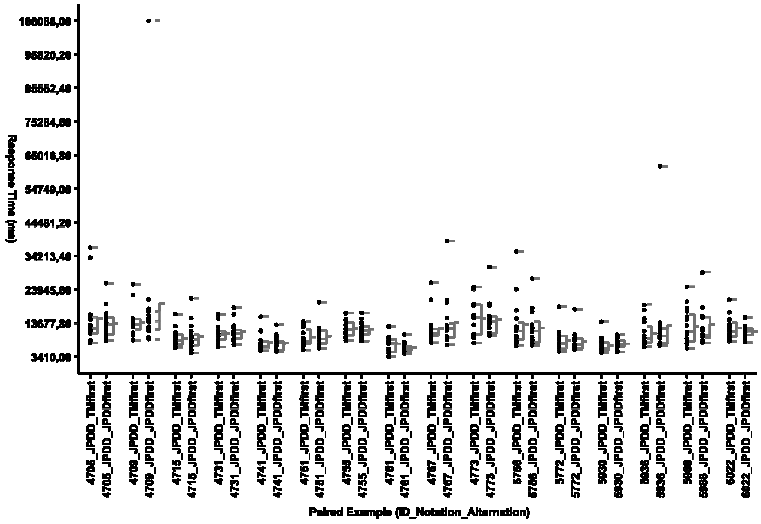


Figure C.2 Response times of task T1 using JPDDs.

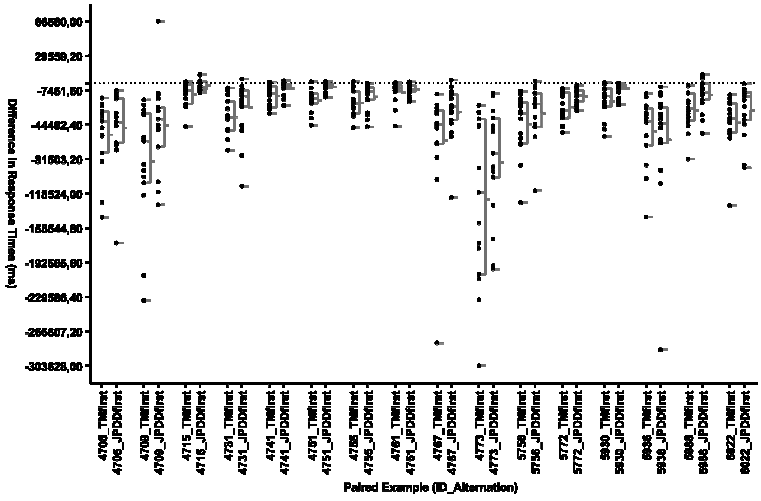


Figure C.3 Differences between JPDDs and Tracematches for task T1.

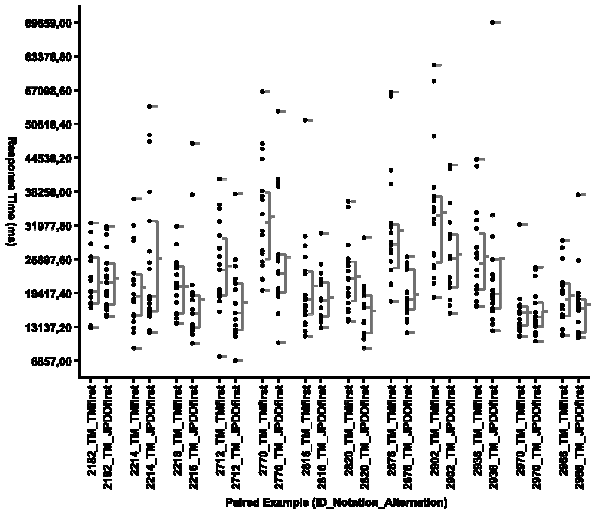


Figure C.4 Response times of task T2 using Tracematches.

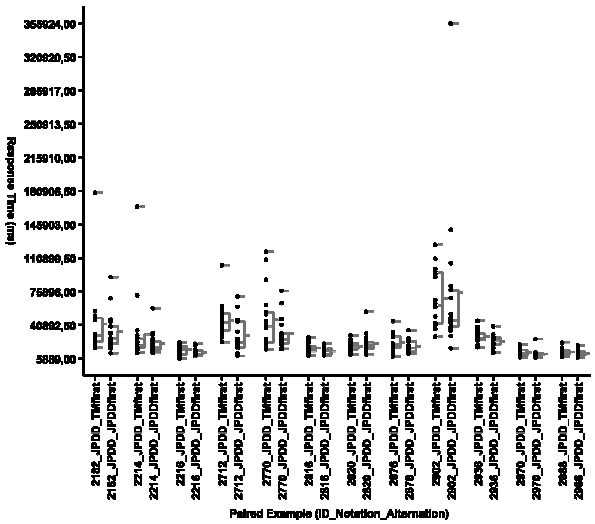


Figure C.5 Response times of task T2 using JPDDs.

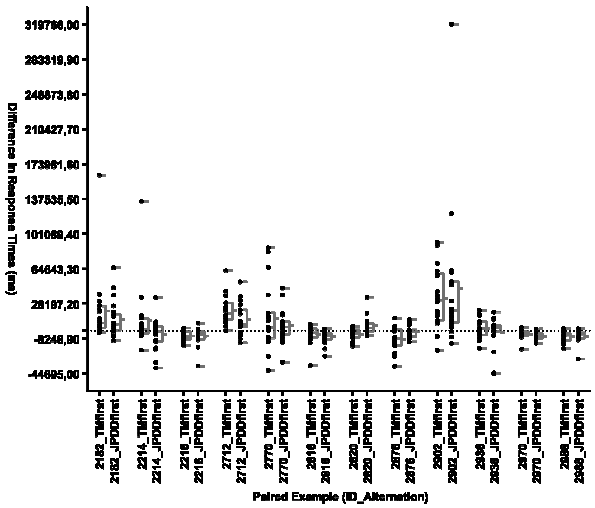


Figure C.6 Differences between JPDDs and Tracematches for task T2.

Table C.1 Estimated marginal means of variable "alternation" for task T1.

T1			confidence interval (99%)	
treatments	marg. mean	std. error	l. bound	u. bound
TMfirst	31,316.97	2,229.03	25,172.90	37,461.04
JPDDfirst	24,769.27	2,158.25	18,820.30	30,718.25
Difference	-6,547.70	3,102.68	-15,099.89	2,004.49

Table C.2 Estimated marginal means of variable "notation" for task T1.

T1			confidence interval (99%)	
treatments	marg. mean	std. error	l. bound	u. bound
TM	44,895.85	2,815.13	37,136.26	52,655.44
JPDD	11,190.40	515.07	9,770.68	12,610.12
Difference	-33,705.45	2,598.83	-40,868.82	-26,542.08

Table C.3 Estimated marginal means of variable "test example" for task T1.

T1			confidence interval (99%)	
treatments	marg. mean	std. error	l. bound	u. bound
4705	40,221.18	3,531.44	30,487.18	49,955.18
4709	49,246.13	4,688.51	36,322.80	62,169.47
4715	12,726.83	1,058.08	9,810.36	15,643.31
4731	26,447.89	2,522.89	19,493.82	33,401.96
4741	12,429.21	1,055.75	9,519.16	15,339.26
4751	15,053.58	1,064.38	12,119.73	17,987.43
4755	20,936.88	1,410.39	17,049.30	24,824.45
4761	10,900.53	1,043.78	8,023.46	13,777.60
4767	36,238.89	4,768.16	23,096.01	49,381.77
4773	67,770.95	7,111.28	48,169.53	87,372.38
5756	31,992.74	2,869.51	24,083.25	39,902.22
5772	18,128.64	1,356.69	14,389.08	21,868.19
5930	13,346.11	1,434.19	9,392.92	17,299.30
5936	39,971.35	5,096.51	25,923.40	54,019.29
5988	23,566.91	1,751.69	18,738.57	28,395.24
6022	29,712.16	2,712.97	22,234.16	37,190.17

Table C.4 Overall marginal means of differences in response time for task T1.

T1	TM vs. JPDD		confidence interval (99%)	
	marg. mean	std. error	l. bound	u. bound
4705	-51,912.33	7,088.40	-71,450.70	-32,373.96
4709	-64,870.58	10,086.01	-92,671.51	-37,069.64
4715	-6,893.42	1,786.50	-11,817.71	-1,969.13
4731	-31,486.02	4,639.15	-44,273.31	-18,698.73
4741	-9,593.24	1,704.22	-14,290.73	-4,895.75
4751	-11,251.77	1,625.14	-15,731.29	-6,772.24
4755	-18,094.70	2,573.33	-25,187.80	-11,001.61
4761	-8,229.84	1,739.81	-13,025.43	-3,434.26
4767	-46,593.03	8,934.71	-71,220.54	-21,965.52
4773	-105,702.90	13,672.30	-143,389.03	-68,016.78
5756	-38,636.60	5,461.65	-53,691.03	-23,582.18
5772	-20,015.22	2,329.56	-26,436.38	-13,594.06
5930	-12,480.16	2,454.35	-19,245.30	-5,715.02
5936	-56,438.12	9,823.38	-83,515.14	-29,361.10
5988	-21,093.51	3,447.42	-30,595.92	-11,591.10
6022	-35,995.75	5,062.20	-49,949.13	-22,042.38

Table C.5 Marginal means of differences in response times for task T1 divided by alternation.

T1	TMfirst TM vs. JPDD		confidence interval (99%)		JPDDfirst TM vs. JPDD		confidence interval (99%)	
	marg. mean	std. error	l. bound	u. bound	marg. mean	std. error	l. bound	u. bound
4705	-55,503.53	10,184.92	-83,577.09	-27,429.97	-48,321.13	9,861.50	-75,503.23	-21,139.02
4709	-84,281.47	14,492.01	-124,227.03	-44,335.90	-45,459.69	14,031.83	-84,136.82	-6,782.56
4715	-12,033.33	2,566.92	-19,108.76	-4,957.90	-1,753.50	2,485.41	-8,604.26	5,097.26
4731	-36,175.53	6,665.73	-54,548.85	-17,802.22	-26,796.50	6,454.06	-44,586.39	-9,006.61
4741	-13,506.60	2,448.70	-20,256.16	-6,757.04	-5,679.88	2,370.94	-12,215.11	855.36
4751	-18,417.47	2,335.08	-24,853.84	-11,981.10	-4,086.06	2,260.93	-10,318.05	2,145.93
4755	-21,267.47	3,697.47	-31,459.13	-11,075.80	-14,921.94	3,580.06	-24,789.97	-5,053.90
4761	-9,542.00	2,499.83	-16,432.50	-2,651.50	-6,917.69	2,420.45	-13,589.39	-245.99
4767	-61,813.93	12,837.77	-97,199.79	-26,428.08	-31,372.13	12,430.12	-65,634.33	2,890.08
4773	-125,104.93	19,644.94	-179,253.96	-70,955.91	-86,300.88	19,021.13	-138,730.44	-33,871.31
5756	-44,474.33	7,847.54	-66,105.17	-22,843.50	-32,798.88	7,598.34	-53,742.85	-11,854.91
5772	-25,842.00	3,347.21	-35,068.20	-16,615.80	-14,188.44	3,240.92	-23,121.67	-5,255.21
5930	-18,990.13	3,526.52	-28,710.58	-9,269.69	-5,970.19	3,414.54	-15,381.97	3,441.59
5936	-51,763.80	14,114.64	-90,669.21	-12,858.40	-61,112.44	13,666.45	-98,782.43	-23,442.44
5988	-29,309.33	4,953.40	-42,962.80	-15,655.86	-12,877.69	4,796.10	-26,097.60	342.23
6022	-42,623.13	7,273.58	-62,671.93	-22,574.33	-29,368.38	7,042.62	-48,780.54	-9,956.21

Table C.6 Estimated marginal means of variable "alternation" for task T2.

T2			confidence interval (99%)	
treatments	marg. mean	std. error	l. bound	u. bound
TMfirst	27,147.61	1,576.60	22,838.32	31,456.90
JPDDfirst	23,068.90	1,622.31	18,634.67	27,503.12
Difference	-4,078.71	2,262.20	-10,261.94	2,104.51

Table C.7 Estimated marginal means of variable "notation" for task T2.

T2			confidence interval (99%)	
treatments	marg. mean	std. error	l. bound	u. bound
TM	22,484.22	775.47	20,364.64	24,603.81
JPDD	27,732.28	1,774.47	22,882.17	32,582.39
Difference	5,248.06	1,543.58	1,029.04	9,467.08

Table C.8 Estimated marginal means of variable "test example" for task T2.

T1			confidence interval (99%)	
treatments	marg. mean	std. error	l. bound	u. bound
2182	29,833.21	2,564.69	22,823.20	36,843.22
2214	24,859.64	2,648.22	17,621.32	32,097.96
2216	16,637.08	855.26	14,299.43	18,974.73
2712	29,408.87	1,884.12	24,259.06	34,558.68
2770	34,375.93	2,205.11	28,348.77	40,403.10
2816	17,469.21	831.15	15,197.44	19,740.98
2820	19,809.03	951.11	17,209.38	22,408.69
2876	22,789.79	1,124.31	19,716.75	25,862.82
2902	50,928.95	5,174.67	36,785.15	65,072.74
2936	26,240.75	1,351.78	22,545.97	29,935.54
2970	13,796.44	627.94	12,080.11	15,512.76
2988	15,150.12	635.38	13,413.44	16,886.80

Table C.9 Overall marginal means of differences in response time for task T2.

T1	TM vs. JPDD		confidence interval (99%)	
	marg. mean	std. error	l. bound	u. bound
2182	16,258.11	5,082.48	2,366.28	30,149.94
2214	3,397.95	4,486.25	-8,864.21	15,660.11
2216	-5,558.44	1,356.48	-9,266.07	-1,850.80
2712	16,739.71	2,699.78	9,360.48	24,118.94
2770	9,067.88	4,733.29	-3,869.51	22,005.26
2816	-4,324.39	1,245.20	-7,727.87	-920.91
2820	968.06	1,375.95	-2,792.80	4,728.92
2876	-4,624.41	1,722.68	-9,332.97	84.15
2902	40,969.01	9,868.34	13,996.10	67,941.91
2936	329.43	2,495.85	-6,492.42	7,151.27
2970	-4,270.20	748.86	-6,317.05	-2,223.35
2988	-5,975.99	1,042.90	-8,826.53	-3,125.45

Table C.10 Marginal means of differences in response times for task T2 divided by alternation.

T2	TMfirst TM vs. JPDD		confidence interval (99%)		JPDDfirst TM vs. JPDD		confidence interval (99%)	
	marg. mean	std. error	l. bound	u. bound	marg. mean	std. error	l. bound	u. bound
2182	20,561.28	7,084.29	1,197.96	39,924.60	11,954.94	7,289.67	-7,969.75	31,879.63
2214	10,896.72	6,253.22	-6,195.06	27,988.51	-4,100.82	6,434.51	-21,688.13	13,486.48
2216	-5,452.17	1,890.75	-10,620.11	-284.22	-5,664.71	1,945.57	-10,982.48	-346.94
2712	21,321.94	3,763.12	11,036.30	31,607.59	12,157.47	3,872.22	1,573.63	22,741.31
2770	12,684.17	6,597.56	-5,348.79	30,717.12	5,451.59	6,788.83	-13,104.17	24,007.34
2816	-3,832.89	1,735.64	-8,576.88	911.10	-4,815.88	1,785.96	-9,697.41	65.64
2820	-3,638.11	1,917.89	-8,880.24	1,604.02	5,574.24	1,973.49	180.13	10,968.34
2876	-8,587.17	2,401.18	-15,150.26	-2,024.08	-661.65	2,470.79	-7,415.01	6,091.72
2902	34,084.78	13,755.13	-3,511.79	71,681.34	47,853.24	14,153.91	9,166.69	86,539.78
2936	2,796.44	3,478.87	-6,712.28	12,305.17	-2,137.59	3,579.73	-11,921.98	7,646.81
2970	-3,312.22	1,043.82	-6,165.26	-459.19	-5,228.18	1,074.08	-8,163.93	-2,292.43
2988	-6,217.39	1,453.67	-10,190.66	-2,244.12	-5,734.59	1,495.81	-9,823.05	-1,646.13

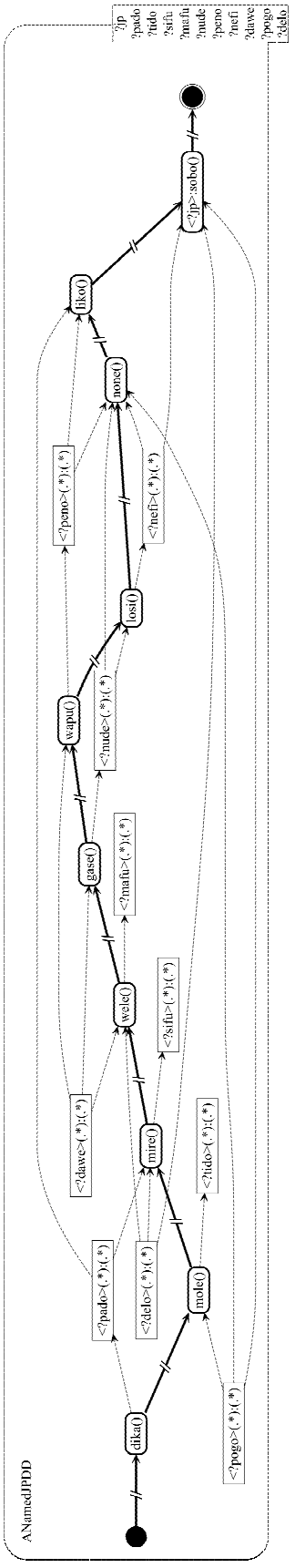


Figure C.7 A test example (of the pretest) presented as JPDD.

Homepages

AJDT
http://www.eclipse.org/ajdt/

Alpha
http://www.st.informatik.tu-darmstadt.de/static/pages/projects/alpha/index.html

Andrew
see CARMA

ANTLR
http://www.antlr.org/

AOSD
http://www.aosd.net/

Aquarium
http://aquarium.rubyforge.org/

Aspect Bench Compiler (abc)
http://www.aspectbench.org/

AspectC++
http://www.aspectc.org/

AspectJ
http://www.eclipse.org/aspectj/

AspectS
http://map.squeak.org/package/e640e9db-2f5f-4890-a142-effebda68748

BPMN
http://www.omg.org/spec/BPMN/

CARMA
http://prog.vub.ac.be/~kgybels/Research/AOP.html

DemeterJ
http://www.ccs.neu.edu/research/demeter/DemeterJava/

Eclipse
http://www.eclipse.org/

EMF
http://www.eclipse.org/emf/

Event-Based AOP (EAOP)
http://www.emn.fr/z-info/eaop/

Firefox
http://www.mozilla.com/

GEF
http://www.eclipse.org/gef/

ImageExport
http://www.se.eecs.uni-kassel.de/~thm/Projects/ImageExport/

JAC
http://jac.objectweb.org/

JAsCo
http://ssel.vub.ac.be/jasco/

JAsCoDT
http://ssel.vub.ac.be/jasco/eclipse_jascodt.html

Java
http://www.java.com/

JPDDs
http://www.dawis.wiwi.uni-due.de/en/research/foci/aosd/jpdds/

M4JPDD
see JPDDs

Magic Draw
http://www.magicdraw.com/

Microsoft Excel
http://office.microsoft.com/en-us/excel/

Monitor-Based AOP
see Event-Based AOP (EAOP)

Motorola WEAVR
http://mypages.iit.edu/~concur/weavr/

MSC
http://www.itu.int/rec/T-REC-Z.120

MySQL
http://www.mysql.com/

OCL
http://www.omg.org/spec/OCL/

Path Expression Pointcuts
no

Perl Aspect
http://search.cpan.org/perldoc?Aspect

Perl Graph
http://search.cpan.org/perldoc?Graph

PROSE
http://prose.ethz.ch/

QVT
http://www.omg.org/spec/QVT/

R
http://www.r-project.org/

RAM
http://www.cs.mcgill.ca/~joerg/SEL/RAM.html

Rational Software Modeler
http://www.ibm.com/software/awdtools/modeler/swmodeler/

SPSS
http://www.spss.com/

Stateful Aspects
see Event-Based AOP (EAOP)

STEAMLOOM
http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp

Theme/UML
http://www.dsg.cs.tcd.ie/aspects/themeUML

Tomcat
http://tomcat.apache.org/

Tracematches
see Aspect Bench Compiler (abc)

Traversal Strategies
see DemeterJ

UML
http://www.omg.org/spec/UML/

UML2
http://www.eclipse.org/uml2/

Date:
November 2010

References

Aksit, M. (ed.) (2003). *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003, Boston, Massachusetts, USA, March 17-21, 2003*, ACM.

Aksit, M., Rensink, A. & Staijen, T. (2009). A graph-transformation-based simulation approach for analysing aspect interference on shared join points, *in* [*Sullivan (2009)*], pp. 39–50.

Al-Mansari, M. & Hanenberg, S. (2006). Path Expression Pointcuts: Abstracting over non-local object relationships in aspect-oriented languages, *in* R. Hirschfeld, A. Polze & R. Kowalczyk (eds), *NODe/GSEM*, Vol. 88 of *Lecture Notes in Informatics*, GI, pp. 81–96.

Alhadidi, D., Boukhtouta, A., Belblidia, N., Debbabi, M. & Bhattacharya, P. (2009). The dataflow pointcut: a formal and practical framework, *in* [*Sullivan (2009)*], pp. 15–26.

Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L. J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. & Tibble, J. (2005). Adding trace matching with free variables to AspectJ, *in* R. E. Johnson & R. P. Gabriel (eds), *OOPSLA*, ACM, pp. 345–364.

Altisen, K., Maraninchi, F. & Stauch, D. (2006). Aspect-oriented programming for reactive systems: Larissa, a proposal in the synchronous framework, *Science of Computer Programming* **63**(3): 297–320.

Arnold, K., Gosling, J. & Holmes, D. (2000). *The Java Programming Language*, 3rd edn, Addison-Wesley, Reading, MA, USA.

Avgustinov, P., Christensen, A. S., Hendren, L. J., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. & Tibble, J. (2005). abc: An extensible AspectJ compiler, *in* [*Mezini & Tarr (2005)*], pp. 87–98.

Avramova, N. (2008). *Improving the usability of an Eclipse-based modeling tool by fully exploiting GEF and ANTLR*, Master's thesis, Institute for Computer Science and Business Information Systems, University of Duisburg-Essen.

Bartelheimer, J. (2006). *Implementation of a modeling tool for join point designation diagrams based on the Eclipse frameworks EMF, GEF, and the UML2-Ecore-Meta-Model*, Master's thesis, Institute for Computer Science and Business Information Systems, University of Duisburg-Essen.

Bergmans, L. (2003). Towards detection of semantic conflicts between crosscutting concerns, *AAOS*. Workshop on Analysis of Aspect-Oriented Software at ECOOP 2003, AAOS 2003, Darmstadt, Germany, July 21, 2003.

Bockisch, C., Haupt, M., Mezini, M. & Ostermann, K. (2004). Virtual machine support for dynamic join points, *in* [*Murphy & Lieberherr (2004)*], pp. 83–92.

Bockisch, C., Mezini, M. & Ostermann, K. (2005). Quantifying over dynamic properties of program execution, *in* R. E. Filman, M. Haupt & R. Hirschfeld (eds), *DAW*, number 05.01, pp. 71–76. Proceedings of the Second Dynamic Aspects Workshop at AOSD 2005, DAW 2005, Chicago, Illinois, USA, March 15, 2005.

Booch, G., Rumbaugh, J. & Jacobson, I. (1998). *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA, USA.

Bortz, J. (1999). *Statistik für Sozialwissenschaftler*, 5th edn, Springer, Heidelberg, Germany.

Box, G. E. P. (1954). Some theorems on quadratic forms applied in the study of analysis of variance problems, ii. effects of inequality of variance and of correlation between errors in the two-way classification, *The Annals of Mathematical Statistics* **25**(3): 484–498.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, Chichester, UK.

Clarke, S. & Baniassad, E. (2005). *Aspect-Oriented Analysis and Design*, Addison-Wesley Professional, Boston, MA, USA.

Clocksin, W. F. & Mellish, C. S. (2003). *Programming in PROLOG: Using the ISO Standard*, 5th edn, Springer, Secaucus, NJ, USA.

Cottenier, T., van den Berg, A. & Elrad, T. (2007a). Joinpoint inference from behavioral specification to implementation, *in* E. Ernst (ed.), *ECOOP*, Vol. 4609 of *Lecture Notes in Computer Science*, Springer, pp. 476–500.

Cottenier, T., van den Berg, A. & Elrad, T. (2007b). Motorola WEAVR: Aspect orientation and model-driven engineering, *Journal of Object Technology* **6**(7): 51–88.

DeMarco, T. (1979). *Structured Analysis and System Specification*, Prentice Hall, Upper Saddle River, NJ, USA.

Douence, R., Fradet, P. & Südholt, M. (2002). A framework for the detection and resolution of aspect interactions, *in* D. S. Batory, C. Consel & W. Taha (eds), *GPCE*, Vol. 2487 of *Lecture Notes in Computer Science*, Springer, pp. 173–188.

Douence, R., Fradet, P. & Südholt, M. (2004). Composition, reuse and interaction analysis of stateful aspects, *in* [*Murphy & Lieberherr (2004)*], pp. 141–150.

Douence, R., Motelet, O. & Südholt, M. (2001). A formal definition of crosscuts, *in* [*Yonezawa & Matsuoka (2001)*], pp. 170–186.

Filman, R. E. (ed.) (2006). *Proceedings of the 5th International Conference on Aspect-Oriented Software Development, AOSD 2006, Bonn, Germany, March 20-24, 2006*, ACM.

Filman, R. E., Elrad, T., Clarke, S. & Aksit, M. (eds) (2005). *Aspect-Oriented Software Development*, Addison-Wesley, Boston, MA, USA.

Filman, R. E. & Friedman, D. P. (2005). Aspect-oriented programming is quantification and obliviousness, *in* [*Filman et al. (2005)*], pp. 21–35.

France, R. B., Ray, I., Georg, G. & Ghosh, S. (2004). Aspect-oriented approach to early design modelling, *IEE Proceedings - Software* **151**(4): 173–186.

Gamma, E., Helm, R., Johnson, R. E. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, USA.

Gansner, E. R. & North, S. C. (2000). An open graph visualization system and its applications to software engineering, *Software — Practice & Experience* **30**(11): 1203–1233.

Geisser, S. & Greenhouse, S. W. (1958). An extension of box's results on the use of the *f* distribution in multivariate analysis, *The Annals of Mathematical Statistics* **29**(3): 885–891.

Gene V. Glass, P. D. P. & Sanders, J. R. (1972). Consequences of failure to meet assumptions underlying the fixed effects analyses of variance and covariance, *Review of Educational Research* **42**(3): 237–288.

Grønmo, R., Sørensen, F., Møller-Pedersen, B. & Krogdahl, S. (2008). A semantics-based aspect language for interactions with the arbitrary events symbol, *in* I. Schieferdecker & A. Hartman (eds), *ECMDA-FA*, Vol. 5095 of *Lecture Notes in Computer Science*, Springer, pp. 262–277.

Gybels, K. & Brichau, J. (2003). Arranging language features for more robust pattern-based crosscuts, *in* [*Aksit (2003)*], pp. 60–69.

Hakala, T., Nykyri, P. & Sajaniemi, J. (2006). An experiment on the effects of program code highlighting on visual search for local patterns, *in* P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (eds), *18th Annual Workshop of the Psychology of Programming Interest Group (PPIG), Brighton, UK, September 7-8, 2006*, pp. 38–52.

Hanenberg, S. (2006). *Design Dimensions of Aspect-Oriented Systems*, PhD thesis, Institute for Computer Science and Business Information Systems, University of Duisburg-Essen.

Hanenberg, S., Hirschfeld, R. & Unland, R. (2004). Morphing Aspects: Incompletely woven aspects and continuous weaving, *in* [*Murphy & Lieberherr (2004)*], pp. 46–55.

Hanenberg, S., Stein, D. & Unland, R. (2007). From aspect-oriented design to aspect-oriented programs: Tool-supported translation of JPDDs into code, *in* B. M. Barry & O. de Moor (eds), *AOSD*, ACM, pp. 49–62.

Harel, D. (1987). Statecharts: A visual formalism for complex systems, *Sci. Comput. Program.* **8**(3): 231–274.

Hietaniemi, J. (2009). Perl Graph. http://search.cpan.org/ jhi/Graph-0.91/lib/Graph.pod

Hilsdale, E. & Hugunin, J. (2004). Advice weaving in AspectJ, *in* [*Murphy & Lieberherr (2004)*], pp. 26–35.

Hirschfeld, R. (2002). AspectS: Aspect-oriented programming with Squeak, *in* M. Aksit, M. Mezini & R. Unland (eds), *NetObjectDays*, Vol. 2591 of *Lecture Notes in Computer Science*, Springer, pp. 216–232.

Hopcroft, J. E., Motwani, R. & Ullman, J. D. (2007). *Introduction to Automata Theory, Languages, and Computation*, 3rd edn, Addison-Wesley Longman, Boston, MA, USA.

Huynh, H. & Feldt, L. S. (1976). Estimation of the box correction for degrees of freedom from sample data in randomized block and split-plot designs, *Journal of Educational Statistics* **1**(1): 69–82.

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*, The Institute of Electrical and Electronics Engineers, New York, NY, USA.

ISO (1998). *ISOIEC 14882:1998: Programming languages — C++*, International Organization for Standardization, Geneva, Switzerland.

ITU (1999). *ITU Recommendation Z.120: Message Sequence Charts (MSC)*, International Telecommunication Union, Geneva, Switzerland.

Jacobson, I. & Ng, P.-W. (2004). *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley Professional, Boston, MA, USA.

Kandé, M. M., Kienzle, J. & Strohmeier, A. (2002). From AOP to UML - a Bottom-Up Approach, *Workshop on Aspect-Oriented Modeling with UML (AOM) at AOSD 2002, Enschede, The Netherlands, April 22, 2002*.

Katara, M. & Katz, S. (2003). Architectural views of aspects, *in* [*Aksit (2003)*], pp. 1–10.

Kellens, A., Mens, K., Brichau, J. & Gybels, K. (2006). Managing the evolution of aspect-oriented software with model-based pointcuts, *in* D. Thomas (ed.), *ECOOP*, Vol. 4067 of *Lecture Notes in Computer Science*, Springer, pp. 501–525.

Kennedy, A., Eilam, R. & Grünauer, M. (2009). Perl Aspect. http://search.cpan.org/ adamk/Aspect-0.21/lib/Aspect.pm

Kiczales, G. (ed.) (2002). *Proceedings of the 1st International Conference on Aspect-Oriented Software Development, AOSD 2002, University of Twente, Enschede, The Netherlands, April 22-26, 2002*, ACM.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001). An overview of AspectJ, *in* J. L. Knudsen (ed.), *ECOOP*, Vol. 2072 of *Lecture Notes in Computer Science*, Springer, pp. 327–353.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. & Irwin, J. (1997). Aspect-oriented programming, *in* M. Aksit & S. Matsuoka (eds), *ECOOP*, Vol. 1241 of *Lecture Notes in Computer Science*, Springer, pp. 220–242.

Kienzle, J., Abed, W. A. & Klein, J. (2009). Aspect-oriented multi-view modeling, *in* [*Sullivan (2009)*], pp. 87–98.

Klein, J., Fleurey, F. & Jézéquel, J.-M. (2007). Weaving multiple aspects in sequence diagrams, in [*Rashid & Aksit (2007)*], pp. 167–199.

Klein, J., Hélouët, L. & Jézéquel, J.-M. (2006). Semantic-based weaving of scenarios, *in* [*Filman (2006)*], pp. 27–38.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*, Manning Publications, Greenwich, CT, USA.

Lämmel, R. & Schutter, K. D. (2005). What does aspect-oriented programming mean to Cobol?, *in* [*Mezini & Tarr (2005)*], pp. 99–110.

Lesiecki, N. (2005a). AOP@Work: Enhance design patterns with AspectJ - Part 1. http://www.ibm.com/developerworks/library/j-aopwork5/

Lesiecki, N. (2005b). Test flexibly with AspectJ and mock objects. http://www.ibm.com/developerworks/library/j-aspectj2/

Levene, H. (1960). Robust tests for equality of variances, *Contributions to probability and statistics: Essays in honor of Harold Hotelling* pp. 278–292.

Lieberherr, K. J. (1995). *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing, Boston, MA, USA.

Mahoney, M., Bader, A., Elrad, T. & Aldawud, O. (2004). Using aspects to abstract and modularize statecharts, *Workshop on Aspect-Oriented Modeling (AOM) at UML 2004, Lisbon, Portugal, September 11, 2004*.

Masuhara, H. & Kawauchi, K. (2003). Dataflow pointcut in aspect-oriented programming, *in* A. Ohori (ed.), *APLAS*, Vol. 2895 of *Lecture Notes in Computer Science*, Springer, pp. 105–121.

Masuhara, H., Kiczales, G. & Dutchyn, C. (2003). A compilation and optimization model for aspect-oriented programs, *in* G. Hedin (ed.), *CC*, Vol. 2622 of *Lecture Notes in Computer Science*, Springer, pp. 46–60.

Mauchly, J. W. (1940). Significance test for sphericity of a normal n-variate distribution, *The Annals of Mathematical Statistics* **11**(2): 204–209.

Mens, K., Kellens, A., Pluquet, F. & Wuyts, R. (2006). Co-evolving code and design with intensional views: A case study, *Computer Languages, Systems & Structures* **32**(2-3): 140–156.

Mezini, M. & Tarr, P. L. (eds) (2005). *Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD 2005, Chicago, Illinois, USA, March 14-18, 2005*, ACM.

Miara, R. J., Musselman, J. A., Navarro, J. A. & Shneiderman, B. (1983). Program indentation and comprehensibility, *Communications of the ACM* **26**(11): 861–867.

Murphy, G. C. & Lieberherr, K. J. (eds) (2004). *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development, AOSD 2004, Lancaster, UK, March 22-24, 2004*, ACM.

Mussbacher, G., Whittle, J. & Amyot, D. (2008). Towards sematic-based aspect interaction detection, *NFPinDSML*. Workshop on Non-functional System Properties in Domain Specific Modeling Languages at MODELS 2008, NFPinDSML 2008, Toulouse France, September 28, 2008.

Nishizawa, M., Chiba, S. & Tatsubori, M. (2004). Remote Pointcut: A language construct for distributed AOP, *in* [*Murphy & Lieberherr (2004)*], pp. 7–15.

OMG (2009). *Business Process Model and Notation (BPMN), Specification Version 1.2*, Object Management Group, Needham, MA, USA.

Ostermann, K., Mezini, M. & Bockisch, C. (2005). Expressive pointcuts for increased modularity, *in* A. P. Black (ed.), *ECOOP*, Vol. 3586 of *Lecture Notes in Computer Science*, Springer, pp. 214–240.

Pawlak, R., Seinturier, L., Duchien, L. & Florin, G. (2001). JAC: A flexible solution for aspect-oriented programming in Java, *in* [*Yonezawa & Matsuoka (2001)*], pp. 1–24.

Popovici, A., Alonso, G. & Gross, T. R. (2003). Just-in-time aspects: Efficient dynamic weaving for Java, *in* [*Aksit (2003)*], pp. 100–109.

Popovici, A., Gross, T. R. & Alonso, G. (2002). Dynamic weaving for aspect-oriented programming, *in* [*Kiczales (2002)*], pp. 141–147.

Purchase, H. C., Cohen, R. F. & James, M. I. (1997). An experimental study of the basis for graph drawing algorithms, *ACM Journal of Experimental Algorithmics* **2**: 4.

Rashid, A. & Aksit, M. (eds) (2007). *Transactions on Aspect-Oriented Software Development III*, Vol. 4620 of *Lecture Notes in Computer Science*, Springer.

Rashid, A. & Chitchyan, R. (2003). Persistence as an aspect, *in* [*Aksit (2003)*], pp. 120–129.

Shapiro, S. S. & Wilk, M. B. (1965). An analysis of variance test for normality (for complete samples), *Biometrika* **52**(3-4): 591–611.

Soares, S., Laureano, E. & Borba, P. (2002). Implementing distribution and persistence aspects with AspectJ, *in* M. Ibrahim & S. Matsuoka (eds), *OOPSLA*, ACM, pp. 174–190.

Spinczyk, O., Gal, A. & Schröder-Preikschat, W. (2002). AspectC++: An aspect-oriented extension to C++, *in* J. Noble & J. Potter (eds), *TOOLS Pacific*, Australian Computer Society, pp. 53–60.

Stein, D. & Hanenberg, S. (2008). M4JPDD: Tool-support for modeling Join Point Designation Diagrams, Demonstration at AOSD 2008.

Stein, D., Hanenberg, S. & Unland, R. (2002). A UML-based aspect-oriented design notation for AspectJ, *in* [*Kiczales (2002)*], pp. 106–112.

Stein, D., Hanenberg, S. & Unland, R. (2004). Query models, *in* T. Baar, A. Strohmeier, A. M. D. Moreira & S. J. Mellor (eds), *UML*, Vol. 3273 of *Lecture Notes in Computer Science*, Springer, pp. 98–112.

Stein, D., Hanenberg, S. & Unland, R. (2005). On relationships between query models, *in* A. Hartman & D. Kreische (eds), *ECMDA-FA*, Vol. 3748 of *Lecture Notes in Computer Science*, Springer, pp. 254–268.

Stein, D., Hanenberg, S. & Unland, R. (2006). Expressing different conceptual models of join point selections in aspect-oriented design, *in* [*Filman (2006)*], pp. 15–26.

Störzer, M. & Graf, J. (2005). Using pointcut delta analysis to support evolution of aspect-oriented software, *ICSM*, IEEE Computer Society, pp. 653–656.

Sullivan, K. J. (ed.) (2009). *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009, Charlottesville, Virginia, USA, March 2-6, 2009*, ACM.

van den Berg, K., Conejero, J. M. & Hernández, J. (2007). Analysis of crosscutting in early software development phases based on traceability, in [*Rashid & Aksit (2007)*], pp. 73–104.

Vanderperren, W. & Suvee, D. (2004). Optimizing JAsCo dynamic AOP through HotSwap and Jutta, *in* R. E. Filman, M. Haupt, K. Mehner & M. Mezini (eds), *DAW*, number 04.01, pp. 120–134. Proceedings of the Dynamic Aspects Workshop at AOSD 2004, DAW 2004, Lancaster, UK, March 23, 2004.

Vanderperren, W., Suvée, D., Cibrán, M. A. & Fraine, B. D. (2005). Stateful Aspects in JAsCo, *in* T. Gschwind, U. Aßmann & O. Nierstrasz (eds), *Software Composition*, Vol. 3628 of *Lecture Notes in Computer Science*, Springer, pp. 167–181.

Walker, R. J. & Viggers, K. (2004). Implementing protocols via Declarative Event Patterns, *in* R. N. Taylor & M. B. Dwyer (eds), *SIGSOFT FSE*, ACM, pp. 159–169.

Wampler, D. (2008). Aquarium: AOP in Ruby, *in* W. Joosen (ed.), *AOSD Industry Track*.

Warmer, J. & Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*, Addison-Wesley, Boston, MA, USA.

Whittle, J., Jayaraman, P. K., Elkhodary, A. M., Moreira, A. & Araújo, J. (2009). MATA: A unified approach for composing UML aspect models based on graph transformation, *Transactions on Aspect-Oriented Software Development* **6**: 191–237.

Yagi, I., Takata, Y. & Seki, H. (2007). A labeled transition model A-LTS for history-based aspect weaving and its expressive power, *IEICE Transactions* **90-D**(5): 799–807.

Yonezawa, A. & Matsuoka, S. (eds) (2001). *Proceedings of Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, REFLECTION 2001, Kyoto, Japan, September 25-28, 2001*, Vol. 2192 of *Lecture Notes in Computer Science*, Springer.

Zhang, G., Hölzl, M. M. & Knapp, A. (2007). Enhancing UML state machines with aspects, *in* G. Engels, B. Opdyke, D. C. Schmidt & F. Weil (eds), *MoDELS*, Vol. 4735 of *Lecture Notes in Computer Science*, Springer, pp. 529–543.

Zloof, M. M. (1977). Query-by-example: A data base language, *IBM Systems Journal* **16**(4): 324–343.