Otto-von-Guericke-Universität Magdeburg

School of Computer Science
Department of Technical and Business Information Systems

Diplomarbeit

Aspect-Oriented Refactoring of Berkeley DB

Author:

Christian Kästner

February 7, 2007

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
University of Magdeburg
School of Computer Science
Department of Technical and Business Information Systems
P.O. Box 4120, D–39016 Magdeburg, Germany

Prof. Don Batory
The University of Texas at Austin
Department of Computer Sciences
2.124 Taylor Hall, Austin, Texas 78712, USA

Kästner, Christian:
Aspect-Oriented Refactoring of Berkeley DB
Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2007.

Contents

List of Figures

List of Tables

List of Abbreviations

ACID	Atomicity, Consistency, Isolation, and Durability
AHEAD	Algebraic Hierarchical Equations for Application Design
AJDT	AspectJ Development Tools
AOP	Aspect-Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
ECR	Extract Call Refinement
EIMR	Extract Inner Method Refinement
ESMR	Extract Simple Method Refinement
FOP	Feature-Oriented Programming
I/O	Input/Output
IDE	Integrated Development Environment
IN	Internal Node
ITMD	Inter-Type Member Declaration
J2EE	Java 2 Platform, Enterprise Edition
JCA	Java Connector Architecture
JE	Java edition
JMX	Java Management Extensions
JRE	Java Runtime Environment
LN	Leaf Node

LOC Lines of Code

OOP Object-Oriented Programming

SJP Shared Join Point

SPL Software Product Line

Chapter 1

Introduction

In this thesis we describe the refactoring of the embedded database engine Berkeley DB into features using *Aspect-Oriented Programming (AOP)* techniques and present the insights gained.

Features are increments in program functionality (Batory et al., 2004). We refactored Berkeley DB into features to create a configurable database engine where the user can decide which features to use. For example, in a single threaded embedded system the developer can have a stripped-down version of the database engine without transactions support, which is smaller and faster than the original version. The benefits of configurable embedded database systems that can be tailored to the users need have been shown in various publications (e.g., Batory, 1986; Tesanovic et al., 2004; Leich et al., 2005c). Applying this refactoring, a family of related programs can be configured from a set of features (Prehofer, 1997; Batory et al., 2004; Griss, 2000). Additionally, refactoring a legacy application into features promises improvements in source code quality, and thus reliability and maintainability, as it separates concerns and encapsulates them in features (Parnas, 1972; Batory et al., 2003; Tarr et al., 1999).

Features are the primary subject in the research area of *Feature-Oriented Programming (FOP)* and *Software Product Line (SPL)* technologies. However, there are various approaches and languages that can be used to implement features. In recent research, it was suggested to use AOP. First results appeared promising, for example Godil and Jacobsen (2005), Tesanovic et al. (2004), and Eibauer (2006) refactored embedded database engines into optional features, and Colyer et al. (2004), Zhang and Jacobsen (2003), and Hunleth and Cytron (2002) refactored middleware to improve customizability. Refactoring of a legacy software into features is also known as *feature-oriented refactoring* (Liu et al., 2006), *horizontal decomposition* (Zhang and Jacobsen, 2003, 2004; Godil and Jacobsen, 2005), *feature-driven product line engineering* (Griss, 2000), or simply *separation of concerns* based on features (Hunleth and Cytron, 2002; Colyer et al., 2004).

Our goal in this thesis is to *evaluate empirically the usability of AOP for feature-oriented refactoring*. We refactor a large real-world application using the aspect-oriented language AspectJ. We describe the refactorings and analyze the process and the resulting source code. In our analysis we focus on answering the following questions from both the feature

perspective and the AOP perspective:

- **Feature with AspectJ:** AspectJ has already been used in some case studies to refactor features. However, these case studies have largely ignored the following questions: *Is AspectJ suited to implement (large-scale) features? What kind of infrastructure is needed and what is provided? Are the language constructs sufficient? Is the resulting feature code readable and maintainable?*

- **Solutions to known Problems:** With features there are known problems. One of the most important and most frequently discussed problem is the Feature Optionality Problem when two optional features interact (Prehofer, 1997; Liu et al., 2005, 2006). *Do such problems occur in Berkeley DB? How are these problems dealt with in AspectJ and does the language provide any solutions superior to existing approaches?*

- **OOP-to-AOP Refactorings:** Recent AOP publications suggested refactorings from object-oriented code to aspects. For our case study we need such refactorings and can give further insight on their usability. *Are these refactorings useful and sufficient?*

- **Ordered Weaving:** Recent research has reasoned that an ordered weaving in AspectJ would reduce complexity (Apel and Liu, 2006), resolve logical problems (Forster and Steimann, 2006), and avoid problems with accidental weaving (Lopez-Herrejon et al., 2006; McEachen and Alexander, 2005). An open question is: *What is the practically significance of ordered weaving?*

- **Stability and Evolution:** The fragility of AOP with respect to subsequent changes, especially of pointcuts in AspectJ, is a known problem in the AOP community (e.g., Koppen and Störzer, 2004; Stoerzer and Graf, 2005; Gybels and Brichau, 2003). In this thesis we analyze how this problem affects a real-world project and how it can be dealt with. By refactoring features one-by-one, we also have a kind of evolution, so that we can observe such fragility problems directly in this case study. We focus on the questions: *Do fragility problems occur in a real-world project? Do they hinder the evolution of the project? Which solutions can be used to avoid this problem?*

- **Expressiveness of AspectJ Language Constructs:** AspectJ was designed as a language proposal for the empirical assessment of AOP (Kiczales et al., 2001). Based on this case study we can give further insight to their question: *Which language constructs are used and how?* It also indicates which limitations exist and *which language constructs are still needed*.

- **Scaling AOP:** The issue of scale is commonly not dealt with in AOP projects. All case studies are very small in terms of *Lines of Code (LOC)*, except for some large projects where the number of features is very small (cf. related work in Chapter 8). In this case study we have the opportunity to observe *how AspectJ scales* to describe small to large features and how *understandability and maintainability* of an AOP project scales as many aspects are created.

1.1 Case Study

In this thesis we conduct a case study in which we refactor the embedded database engine Berkeley DB into features. To implement features we use the aspect-oriented language AspectJ.

1.1.1 Scope and Delimitation

For the case study in this thesis we chose to *refactor an existing application* over writing a new application. This gives us the chance to experiment on a large scale and eliminates the possibility to write a case study that is biased by choosing an implementation that suits or does not suit the needs of AOP. Instead we refactor a well known existing application that was designed without the use of aspect-oriented techniques. Furthermore, refactoring an existing application allows to compare results.

We furthermore chose the aspect-oriented language *AspectJ* over other aspect-oriented languages. Because aspect-oriented languages differ significantly, some results may not be transferable to other aspect-oriented languages. However, AspectJ is one of the most popular and accepted aspect-oriented languages. There is a good tool support for it, and AspectJ has matured over several releases. Finally, it was also used in all case studies shown above and in most research on refactoring and evolution, which helps comparing results.

Finally, the case study must not be understood on the background of feature-oriented refactoring and not on the background of modularizing homogeneous crosscutting concerns. Most prior work focuses on the improvement of a program by modularizing (small, homogeneous) crosscutting concerns into technical aspects. This emphasizes the limitations of object-oriented languages that can be addressed with AOP specifically. These technical aspects are usually not optional and not used to configure a program, though this might be a side effect in some cases. Instead, in this thesis, we focus on implementing features that encapsulate increments in functionality, which is in line with prior work on FOP and SPLs. Therefore, we do not follow an Aspect Mining approach, that usually is used to find replicated or cloned code, but instead searches for design decisions and ‘concerns’ implementing increments in functionality given by the domain.

This distinction between typical technical aspects and features is also a difference to most previous studies which claimed to refactor features or concerns. These studies (e.g., Zhang and Jacobsen, 2003; Eibauer, 2006) refactored only selected features (cf. Section 8 for more related case studies). Usually, they still focus on crosscutting concerns and ignore other features and ideas from FOP and SPLs. Additionally, our focus on scale, readability, expressiveness and ordering is novel compared to previous studies.

1.1.2 Berkeley DB

As the target application for the refactoring project we have chosen the Java edition of the open source embedded database engine Berkeley DB.

Berkeley DB—since September 2006 also marketed as *Oracle Berkeley DB*—is a light-weight database management system (or more precisely 'storage engine') implemented both in C and in Java. It is broadly used in many projects, both commercial and scientific, the vendor describes its popularity in the data sheet as follows:

> *Berkeley DB is used within a wide variety of open source and proprietary products for telecommunications, networking, storage, security, enterprise infrastructure, financial services and many other markets. There are over 200 million copies of Berkeley DB in deployment. Open source projects that use Berkeley DB include every version of Linux, BSD UNIX, Sendmail, OpenLDAP, Apache, and many other open source projects. Proprietary products that use Berkeley DB include directory servers from Sun and Hitachi; messaging servers from Openwave and LogicaCMG; switches, routers and gateways from Cisco, Motorola, Lucent, and Alcatel; storage products from EMC and HP; security products from RSA Security and Symantec; and Web applications at Amazon.com, Google and AOL.* (Oracle, 2006a)

The *Java edition (JE)* was developed after the C version was already established. It is implemented completely using the Java language and following the object-oriented paradigm. Because the basic architecture was already known from the C version, we believe that the Java edition was developed as a final design, contrasting many other legacy applications that were constantly extended or changed. Important features like transaction management were not added after the design phase, instead Berkeley DB JE had a clear design with all features from the very beginning[1].

Additionally, Berkeley DB JE is designed and implemented in an object-oriented way, e.g., it organizes code into classes and uses inheritance for code reuse. We have chosen Berkeley DB JE because it represents a modern, state of the art application where we can evaluate the advantages of AOP compared to a clean object-oriented design. Specifically, we avoid distorted results that arise from effects specific to procedural programming or weakly designed legacy application.

As an embedded, light-weighted database engine Berkeley DB JE does not support many features known from conventional relational database management systems, like an ad-hoc query language. Nevertheless it still contains many features that appear promising to refactor into optional features. Among them are high concurrency, full ACID transactions, automatic recovery, a high performance no-rewrite storage system, indexes, an in-memory cache and various background threads.

Berkeley DB JE[2] is a mid-sized application of about 84,000 LOC and 300 classes. As a broadly used and commercially supported open source application, it is well documented, especially its usage and interface methods.

[1] Berkeley DB JE introduces some new concepts that are different from the C version. The most important is a different persistence layer using a non-rewrite log instead of a traditional combination of one database file and rotated log files. Also the memory management is very different from the C version. Still the basic concepts were known at design time and not altered afterwards.

[2] For the remainder of this thesis we refer only to the Java edition of Berkeley DB and therefore drop the JE appendix of the name.

1.2 Structure of this Thesis

The remainder of this thesis is structured as follows:

Chapter 2 In Chapter 2 we give some background on the used technology and terminology. First we present the architecture of Berkeley DB. Then, we introduce the concept of Feature-Oriented Programming and Aspect-Oriented Programming and the aspect-oriented language AspectJ. Finally, we give an overview of AHEAD.

Chapter 3 In Chapter 3 we present our refactorings. We start with a description of the infrastructure used, and a list of features we identified in Berkeley DB. Then, we describe the actual refactoring process and which refactorings were applied. Finally, we analyze the resulting features, their size, and their dependencies, so that we can show the feature model.

Chapter 4 In the following Chapters 4–7 we discuss results and insights from the case study. We start by analyzing the expressiveness of the AspectJ language constructs in Chapter 4 and show which language constructs were used and how. We also discuss limitations found during the refactorings and how they can be solved.

Chapter 5 Subsequently, in Chapter 5 we discuss the Feature Optionality Problem known from FOP and its consequences regarding our case study. We compare two approaches to cope with this problem and analyze whether they are practical and whether they scale for large feature models.

Chapter 6 In Chapter 6 we focus on a major difference between FOP and AOP: the ordering of features. We evaluate recent proposals for an ordered weaving based on our refactorings. We focus especially on Shared Join Points and Accidental Weaving.

Chapter 7 The last major discussion focuses the understandability and maintainability of the refactored features in Chapter 7. First, we analyze the features' source code and the AspectJ syntax regarding readability and understandability. Then, we discuss issues of maintainability and evolution based on our experience and review possible suggestions for improvements.

Chapter 8, 9 Finally, we relate our case study to former work in software engineering in Chapter 8, before we conclude the thesis in Chapter 9, put our results in perspective, and list suggestions for further work.

Chapter 2

Background

2.1 Architecture of Berkeley DB

To refactor Berkeley DB, an understanding of its architecture is necessary. Furthermore, Berkeley DB uses some terms differently from common literature on (relational) databases. Therefore, we start with an introduction of the architecture and concepts of Berkeley DB.

We depict the overall architecture of Berkeley DB in Figure 2.1[1]. It consists of five parts: the access methods, the B^+-tree, the concurrency and transaction system, the caching and buffering system and finally the persistence layer. The remainder of this section describes this architecture and how the depicted features integrate in this architecture.

Access Methods. The application that integrates Berkeley DB communicates with the database through various access methods and objects. From the application's perspective, the database system consists of the following parts: the environment, multiple databases, secondary databases, and cursors.

The environment represents an instance of the database system. To use Berkeley DB the application creates an environment handle for a specific directory and sets its global settings, e.g., if the whole system is writable. An environment is a container for multiple databases. A database in Berkeley DB is an entity that can store key-value pairs. Databases are identified by a name and can be compared to tables in common relational databases systems. Databases have *Application Programming Interface (API)* methods to store and retrieve single entries and maintain various per-database settings, e.g., support for duplicate keys. Secondary databases are used to access data from a database with a second key. Therefore, they can be compared to indexes in traditional database systems. Finally, cursors are a mechanism to iterate over databases. They are usually used to access, store or delete more than one key-value pair.

An application uses these objects to access the database to read or write data. They are well documented in manuals and the API documentation of Berkeley DB. Below these

[1]This figure is an extended and more detailed version of the general architecture shown at the Oracle Technology Network (`http://www.oracle.com/technology/products/berkeley-db/je`).

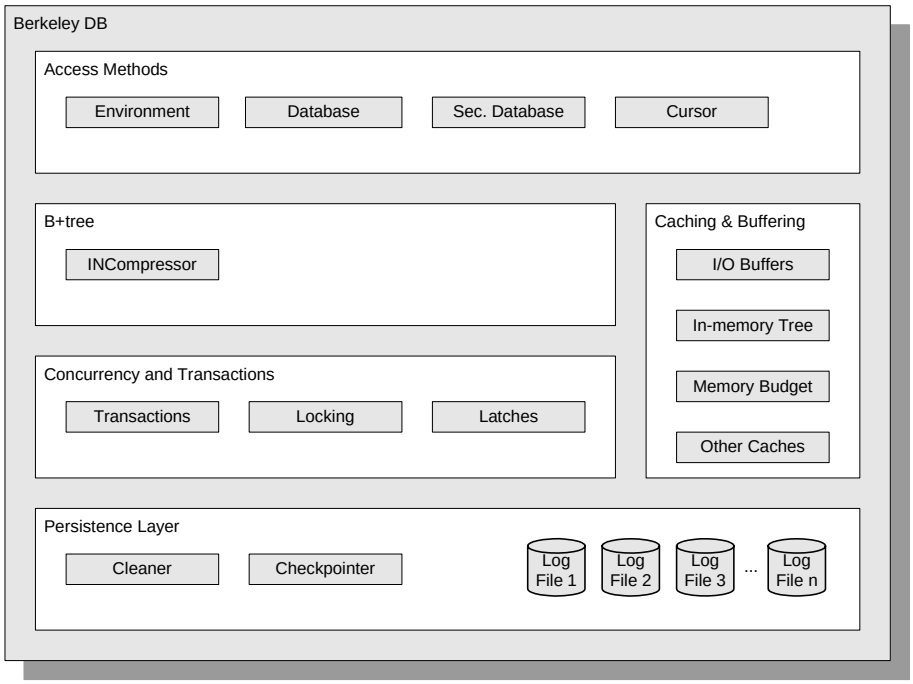


Figure 2.1: Architecture Overview of Berkeley DB.

public interfaces Berkeley DB uses a completely different data structure to store data in a B^+-tree. The mechanisms described above are the ones the user of the database can access.

B^+ Tree. In Berkeley DB all data is stored internally as key-value pairs in one B^+-tree. A B^+-tree is a B-tree where each *Internal Node (IN)* contains only keys, while the *Leaf Nodes (LNs)* contain the data, in this case one individual key-value pair (Comer, 1979). Using this approach a broader tree can be achieved by fitting more keys into one internal node and record level locking is possible by locking only the leaf nodes.

To increase performance internal nodes are not cleaned or deleted when leaf nodes are deleted in Berkeley DB, but only marked. The removal of unnecessary entries in the tree, i.e., checking an underflow condition for INs, is done periodically by a background daemon thread called INCompressor. The INCompressor runs every 5 seconds to perform the necessary clean up operations. As a consequence of this lazy deletion approach, database operations that involve deleting nodes are faster because they do not to deal with the removal of keys from INs.

Concurrency and Transactions. Berkeley DB supports high concurrency for both read and write operations. The application can use any number of threads to access concurrently the database. Additionally, it is even possible for two processes, i.e. two different applications, to share one database, even though only one process can perform write operations. This is uncommon compared to conventional database management systems in

which applications access the database through one specific database process—the database server—that exclusively accesses the database files.

To allow multi-threaded access on the database, Berkeley DB synchronizes most internal operations using a fine-grained self-written synchronization mechanism called *latches*. Latches are created by a synchronization manager, that can be implemented with different techniques, and are used to ensure that only one thread at a time can modify certain internal data. For example, internal operations on tree nodes inside the B^+-tree are usually latched. Berkeley DB has no deadlock detection for latches, therefore the implementation has to avoid deadlock itself by holding latches only for a short time and acquiring and releasing them in a certain sequence.

The transaction system enforces atomicity, consistency, isolation, and durability (ACID) of operations. The ACID requirements are implemented in different ways in the system.

Atomicity is guaranteed by a part called just 'Transactions' in Berkeley DB. In contrast to common terminology the term Transaction in Berkeley DB only refers to the part responsible for atomicity[2]. To achieve atomicity the user can request an transaction object from the database, which he passes to all operations he wants to perform under this transaction, before he calls commit or abort on this transaction object. Atomic transactions in Berkeley DB are optional and can be turned off on both, environment and database level.

Consistency and isolation in Berkeley DB are achieved by the locking subsystem, which locks database entries on record level. As database records are stored only in Leaf Nodes in a B^+-tree, only those need to be locked. For read operations four different isolation levels are available, the highest being serializable isolation. For performance reasons it is possible to lower the isolation level for specific operations to 'repeatable read', 'read committed' or 'read uncommitted'.

There are different types of locker objects that are responsible for acquiring and releasing locks. The used implementation depends on a variety of factors, including whether atomic transactions are used and which isolation level is required. Locking is also optional and can be turned off at environment level. When atomic transactions and locking are turned off dummy locker objects are instantiated with empty lock and release methods.

Finally, durability is implemented on persistence level. When an operation is committed all changed LNs are immediately written to the log and can be recovered in the event of a crash.

To conclude, with a combination of the latching, locking and atomic transaction subsystems an high concurrency system with full ACID compliant transactions was implemented.

Persistence Layer. The persistence layer of Berkeley DB is implemented in an unusual way compared to most other database management systems. Many database management systems and also the C version of Berkeley DB use a database file for all data and rotating log files for emergency recovery. Typically, changes are synchronously written to a log file and are asynchronously written to the database file. Once the database file is updated

[2]To distinguish between these different meanings we use the term 'atomic transactions' for the atomicity subsystem called transactions in Berkeley DB and the term 'transaction system' to talk about the whole ACID transaction system including locking.

the database management system can overwrite or delete the log file unless it is needed for additional backup purposes.

In contrast, Berkeley DB uses log files only. It attaches only new data to the end of the log instead of changing it in a database file. This technique is called a *no-rewrite log* and has various advantages. First, write operations are faster because they need to be written only once. Second, the system does not have to deal with situations where it must replace an entry with a newer, larger one which requires to relocate or to split the entry. Finally, the no-rewrite log approach simplifies backup and restore, because log files can be copied sequentially without having to worry about consistency and restore is already part of the normal operation of the system.

On the other hand, this approach implies two problems that were avoided by traditional approaches. First, log files grow constantly and add even updated entries to the end of the log file. To delete or overwrite older log files is not possible because they still can contain needed entries. Second, a recovery must be run every time the database system is loaded, not only in the case of a crash. As mentioned above only LNs are written synchronously to ensure durability and INs are written asynchronously only when they are removed from the cache to improve performance, because they can be recovered from the LNs. Starting at the end of the log, the recovery process reconstructs the B^+-tree with the latest version of the entries. Consequently, as the log file grows, also the recovery time increases.

The first problem of deleting old log files is tackled by using a sequence of smaller log files and a background process simply called Cleaner. The persistence system uses a sequence number to name and address individual log files. Once the current log file has reached a size limit, the sequence number is increased and a new log file is created. Contrasting log files in traditional database systems, Berkeley DB cannot rotate log files, instead one file is created after another. To free unused disk space, the Cleaner process cycles through the old log files and searches for files with a high number of entries that were deleted or updated in a newer log file. If the number of obsolete entries exceeds a certain limit, by default 50 % of all entries, the Cleaner copies those entries that were not updated to the newest log file. When all entries are moved it deletes the old log file. This way the Cleaner can ensure that the overhead costs of disk space inflicted by the non-rewrite log architecture does not exceed a certain percentage of the database's size.

The second problem of long recovery times is approached with a checkpoint subsystem. To shorten this recovery time, a so called Checkpointer background thread is responsible to write the latest version of all INs to the database in certain intervals, together with checkpoint markers. At each completed checkpoint it is ensured that all changes to the INs, up to the point in time where the checkpoint started, have been saved. Therefore, the Berkeley DB must recover only potentially lost changes after the last completed checkpoint, thus reducing the recovery time.

Using this persistence layer Berkeley DB is able to scale, from small databases, that completely fit into main memory, up to databases that handle hundreds of gigabytes of data (Oracle, 2006b).

Caching & Buffering. To increase performance Berkeley DB uses various caches and buffers. The most important are the in log buffers for read and write operations and the in-memory cache that holds a part of the B^+-tree. Furthermore, there are various minor caching facilities for different purposes, e.g., caching file handles.

Log buffers are used for all *Input/Output (I/O)* operations and by default take 7 % of the overall memory. When a node must be read from the log file, first the buffers are checked if they contain the entry. A log buffer pool holds all the log buffers and maintains statistics to calculate the buffer hit ratio.

Berkeley DB keeps an in-memory B^+-tree that contains parts of the full B^+-tree in the log files as an cache. Some nodes like the root and the nodes for database names are always kept in memory, while other nodes are loaded when needed and kept as long as memory is available. When memory is needed, e.g., to load new parts of the database into memory, some nodes of the in-memory B^+-tree are released and – if changed – written to the log file. The releasing of memory by removing nodes from the in-memory tree is called eviction and done by the Evictor subsystem. Berkeley DB runs the Evictor synchronously before memory is needed for an operation and additionally maintains it as as a background daemon thread that always keeps memory usage below a certain level[3]. The eviction is executed by removing all references to selected nodes, so that the garbage collector of the *Java Runtime Environment (JRE)* can free the memory. To select which nodes should be released, the Evictor uses a Least Recently Used strategy and additionally priorizies LNs on a high level in the tree.

To determine the necessity for eviction Berkeley DB needs information about the current memory usage. However, in the Java environment an application does not have access to information about available memory or memory size of certain objects. Therefore Berkeley DB implements its own memory strategy. It monitors the memory usage by estimating the size of all cached objects to approximate the overall memory usage using the so called memory budget subsystem. With this approximated memory usage Berkeley DB can decide when it needs to release memory by running the Evictor. To calculate the estimated memory usage in the in-memory B^+-tree, every node in the tree is given the ability to calculate its own size based on the size of their content, like keys and values, and based on some constants for the anticipated memory footprint of Java objects. Whenever a node is added, changed or removed the memory budget is informed about this change. The memory usage for other subsystems, e.g., locking, is calculated and updated similarly.

Using the in-memory B^+-tree together with the memory budget and Evictor Berkeley DB can efficiently use the assigned memory to cache database operations and increase performance, even though memory management is not directly supported by the JRE.

[3]Per default the Evictor daemon thread runs on every change of memory usage and evicts nodes until the memory usage is below 95 % of the available memory or that at least 512 kbyte of free memory are available.

2.2 Refactoring

A refactoring is a behavior-preserving program transformation that updates an application's design and underlying source code (Tokuda and Batory, 2001). Usually refactorings are performed to improve the design of an application in some respect, therefore Fowler defines refactorings as following:

> ***Refactoring*** *(noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.* (Fowler, 1999, p. 53)

To refactor an application means to apply a series of refactorings to improve the application, again without changing the observable behavior. In Fowler's definition a refactoring has the aim of making source code easier to understand and maintain. The need for refactorings is usually indicated by 'code smells'. Code smells are observed effects that are commonly regarded as bad design or bad coding style, including duplicated code, overlong methods, large classes, long parameter lists and many more (Fowler, 1999, Chap. 3). For such code smells there are known transformations that can fix these problems, e.g., move duplicated code to a method or split large methods. These transformations are described as refactorings and formalized to assure that they are behavior conserving (Fowler, 1999, Chap. 5ff).

Refactorings are always named and often collected in catalogs for reference. Typically they first contain only an informal step-by-step description, but are often formalized with pre-conditions and exact transformations that ensure their behavior-preserving character. Many modern *Integrated Development Environments (IDEs)* have build in support for refactorings that automate these steps. For example in Eclipse a developer can select from various refactorings and has to specify only few necessary parameters. The IDE then performs the refactoring automatically.

In the following we show the Extract Method refactoring which we use frequently in our case study. The Extract Method refactoring is applied on methods that are too long or hard to understand. It moves some inner statements to a new method and calls the new method from the original position in the long method. Thus, the original method is shorter and the purpose of statement sequence is easier to understand because it is named by the method name. When the statement sequence requires access to local variables, these must be passed as parameters to the new method. In Figure 2.2 we show an example of the Extract Method refactoring. An automated refactoring can do this automatically: after selecting the target statements, it only requests the name of the new method as shown in Figure 2.3. All other values are preselected automatically. Furthermore a tool ensures that the transformation is behavior-preserving, e.g., by ensuring that the new method does not exist already.

Refactorings have several advantages. First, they automate design changes and can thus significantly speed up the process of changing the source code. While error-prone manual changes must be tested, automated refactorings can ensure a correct transformation (Tokuda and Batory, 2001). Second, refactorings avoid the need for overly complex designs, as they make later design changes easy. They also allow engineers to experiment with different

```
1  public class Environment {
2    public void close(boolean force) {
3      prepareShutdownDaemons();
4      //close all databases
5      boolean success = true;
6      for (Database db : openDatabases) {
7        success &= db.close(force);
8      }
9      if (success || force)
10       shutdownDaemons();
11   }
12 }
```

(a) Original Code.

```
1  public class Environment {
2    public void close(boolean force) {
3      prepareShutdownDaemons();
4      boolean success =
           closeDatabases(force);
5      if (closeDatabases(force) || force)
6        shutdownDaemons();
7    }
8
9    private boolean closeDatabases(boolean
         force) {
10     boolean success = true;
11     for (Database db : openDatabases) {
12       success &= db.close(force);
13     }
14     return success;
15   }
16 }
```

(b) Refactored Code.

Figure 2.2: Extract Method Refactoring Example.

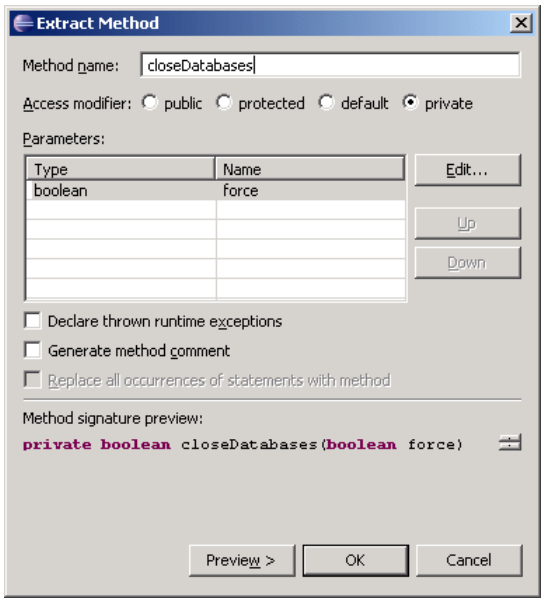


Figure 2.3: Extract Method Refactoring: Eclipse Dialog.

designs and ease evolution. Therefore, they are elemental part of development approaches like *Extreme Programming* (cf. Beck and Andres, 2004).

Note, in this thesis, we often use the term refactoring independently of immediate improvements of the source code. In fact, we will use refactorings in cases where they make the source code quality worse by common standards, e.g., we will use the Extract Method refactorings for single statements or even to create empty methods. We usually use the term as defined by Tokuda and Batory (2001) as a behavior-preserving transformation. We use refactorings as a tool to change source code without judging source code quality.

Finally, Hannemann et al. (2005) argue to use refactorings for intension-preserving transformations instead of behavior-preserving . This means that refactorings are allowed to change the behavior of the application as long as the original intension is maintained. We do not follow this approach, because it is hard to determine the exact intension of legacy code. For example, we could implement the synchronization code differently as long as it still synchronizes the database operations, however such changes would be too dangerous to perform, because we might miss some intended use cases, e.g., use cases that are not yet covered by unit tests. Therefore, we perform only behavior-preserving transformations, because thus we can ensure to also preserve the original intension.

2.3 Feature-Oriented Programming

Feature-Oriented Programming (FOP) is a design and implementation methodology for programs and families of programs. The basic abstractions of FOP are *features*, which are increments in program functionality. In Batory et al. (2004) a feature is defined as a *"product characteristic that is used in distinguishing programs within a family of related programs."*.

The aim of FOP is to create a *Software Product Line (SPL)* out of a *program family*, in which features are the main building blocks that reflect a structure of interest to stakeholders. Products of an SPL are distinguished by their features. Technically, a software product is composed from a base that contains the code shared by all members, and code that implements several features. Features implement program transformations, i.e., when composed with a program, the feature adds and extends functionality. By selecting which features to compose, it is possible to synthesize different programs from the same code basis.

Berkeley DB has several such features, for example, transactions, logging, different I/O operations, or different access methods. There are possible configurations in which Berkeley DB might be composed without any of these features or with a selection of them. For example, a configuration without transactions can be much faster in a single-thread environment.

There are many different ways to implement features, for example, Batory et al. (2004) list layers, feature modules, metaclasses, collaborations, subjects, concerns and aspects. In this thesis, we use aspects in collaboration with classes to implement features.

All features of a domain and their relationship are described in a *feature model* (Kang et al., 1990; Czarnecki and Eisenecker, 2000; Batory, 2005). For example, certain features

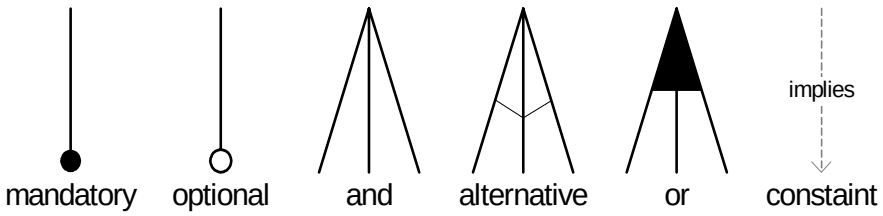


Figure 2.4: Feature Diagram Notations.

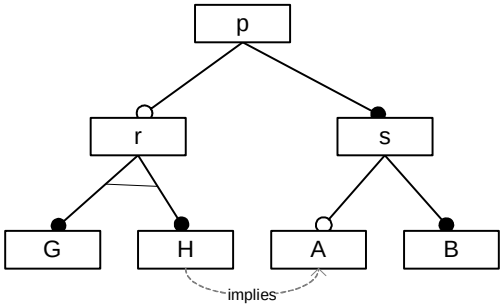


Figure 2.5: Feature Diagram Example (adapted from Batory, 2005).

in a domain might be required, while others are optional or mutual exclusive. Furthermore, some features might require other features and can only be used in combination. Technically, a feature model describes all configurations possible for an SPL of a domain. It describes which features can be applied to the base code and which constraints are present.

Current methodologies organize feature models as trees, which can be visualized as *feature diagram* (Kang et al., 1990; Czarnecki and Eisenecker, 2000). In the frequently used notation as depicted in Figure 2.4, the relationship between a parent feature and a child feature is either mandatory or optional. Furthermore, when a feature has multiple child features there are three different relationship types: (1) *and:* all child features must be selected, (2) *alternative:* one one child feature can be selected, and (3) *or:* one or more child features can be selected. Additionally, we introduce an additional notation for constraints that cannot be described in the tree structure. There are various graphical tools to model these feature diagrams like AmiEddi[4], FeatureIDE (Leich et al., 2005a), FeaturePlugin (Antkiewicz and Czarnecki, 2004), or XFeature[5].

In Figure 2.5 we show an example of a feature diagram with this notation. It consists of the base code *p* and two child features *r* and *s*, where *s* is mandatory and *r* is optional. The feature *r* has again two child features, where exactly one must be selected. The feature *s* has two child features. Additionally, *A* must be selected when *H* is selected, because of a constraint. One valid configuration for this example is *p, s, B*, another is *p, r, s, H, A, B*.

[4] http://www.generative-programming.org/
[5] http://www.pnp-software.com/XFeature/

```
1 p: [r] s;
2 r: G | H;
3 s: [A] B;
4 %%
5 H implies A;
```

Figure 2.6: *guidsl* Grammar Example.

Feature models can also be described with *feature grammars* (de Jonge and Visser, 2002; Czarnecki et al., 2005; Batory, 2005) or *logic programming* (e.g., Mannion, 2002). Feature grammars are especially useful to describe models in a computer readable format and to specify constraints between features that can not be modeled with the relationships of feature diagrams. In this thesis we use the *guidsl* grammar as introduced by Batory (2005). The *guidsl* grammar is an iterative tree grammar, where features are defined in the form *'A: B C; B: D E;'*, which means that feature *A* has the child features *B* and *C*, and *B* itself has the child features *D* and *E*. Optional features are expressed with square brackets (e.g., *'A: [B] [C];'*), repetitions with an asterisk or a plus sign (e.g., *'A: B* C+;'*) and alternative features with the pipe symbol (e.g., *'A: B | C;'*). *Or* relationships can be expressed with a combination of the above (e.g., *'A: [B]+; B: C | D | E;'*). Additionally, the *guidsl* grammar can express additional constrains, usually inclusions or exclusions ('choosing feature A automatically includes a given feature list'). Such constrains are described as logical expressions like *'A implies B or C or D;'* or *'(A or B) and C implies D;'*. For more details on the *guidsl* grammar see Batory (2005).

In Figure 2.6 we show the *guidsl* grammar for the above feature diagram example. It describes exactly the same model, just with a different representation. In large feature models, feature diagrams tend to become complex and hard to read. The grammar representation helps to read such feature models and is easier to handle with a tool like *guidsl*.

2.4 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is a programming paradigm that specializes on modularizing crosscutting concerns. It builds on previous technologies including procedural programming and object-oriented programming, that have already made significant improvements in software modularity (Kiczales et al., 2001). It addresses specifically those concerns that cut across whole systems, which traditional approaches are unable to modularize.

In traditional procedural or object-oriented systems, certain concerns called *crosscutting concerns* like logging, caching, synchronization, or policy enforcement are spread throughout the program. Source code to implement these concerns, e.g., calls to logging methods, are repeated *('scattered')* in various methods of the program. It is not possible to encapsulate them in one coherent module, instead these calls are mixed *('tangled')* with other concerns in other modules.

AOP provides means to encapsulate these crosscutting concerns in so called *aspects*.

An aspect is a module that implements a crosscutting concern. Aspects are usually implemented in an *aspect-oriented language*, although the implementations differ significantly between different languages.

Central to most aspect-oriented languages is a concept called *quantification*, that allows aspects to make extensions in the form *"In programs P, whenever condition C arises, perform action A"* (Filman and Friedman, 2005). The condition is usually based on a pointcut language that matches well-defined points in the execution of an program, called *join points*. Every aspect-oriented language provides its own join point model, but in most languages possible join points are method executions, class initializations or method calls. Some languages also provide advanced join points, e.g., throwing an exception. *Pointcuts* are used to specify one or more join points. There are many different ways how pointcuts can be defined, most languages support static conditions like "whenever method M is called", others support dynamic conditions like "when the call stack has reached a size over 20" or "when the password was entered incorrectly for 5 times". When the condition is fulfilled, i.e., the pointcut matches the current join point, *advice* is executed to perform the action *A*.

Another characteristic often associated with AOP is *obliviousness* which describes that the program can be developed oblivious of later application of aspects. This means that the program's code does not have to be prepared for the later extension by aspects. Filman and Friedman (2005) describe the goal of obliviousness like this *"Just program like you always do, and we'll be able to add the aspects later"*. Obliviousness is controversial, because it breaks encapsulation and can lead to incorrect extensions (e.g., Clifton and Leavens, 2003; Koppen and Störzer, 2004; Aldrich, 2004; Murphy and Schwanninger, 2006).

A concrete and generally accepted definition of AOP is not available. AOP approaches are diverse and which languages are aspect-oriented languages and which are not is frequently discussed. However, the suggestion *"Aspect-Oriented Programming is Quantification and Obliviousness"* by Filman and Friedman (2005) is often agreed on as definition. Alternatively, AOP is sometimes defined simply by its aim, *"modularizing crosscutting concerns"* (cf. Steimann, 2006).

AOP has gained popularity in both research and industry (cf. Sabbah, 2004; Steimann, 2006). There are many different aspect-oriented languages and aspect-oriented extensions for almost every popular programming language.

2.5 AspectJ

AspectJ is a prominent representation of an aspect-oriented languages. It is an AOP extension to the Java programming language and comes with a fully functional compiler and broad tool support, for example with an advanced plug-in for the Eclipse development environment (Kiczales et al., 2001; Clement et al., 2003). AspectJ has become very popular and is the flagship implementation of the AOP community.

AspectJ is an *Upward compatible* and *Platform compatible* aspect-oriented language extension to Java (Kiczales et al., 2001). This means that every Java program is also a valid AspectJ program, and every AspectJ program can be compiled to standard Java byte-code,

that is executable in standard Java virtual machines. Furthermore, AspectJ is designed to be *Tool compatible* and *Programmer compatible*, meaning that it should be possible to extend existing Java tools like IDEs or documentation tools with AspectJ in a natural way. AspectJ should also *"feel like a natural extension of programming with Java"* (Kiczales et al., 2001).

The AspectJ language has matured in various releases. There are also different compilers available, with minor differences in language semantics. In this thesis we use AspectJ in the current version 1.5.3 and with the *ajc* compiler[6].

Dynamic Crosscutting. The AspectJ join point model provides join points at nodes in the runtime object call graph of the program (Kiczales et al., 2001). Thus, AspectJ can extend join points at which an object sends or receives a method call, at which a member variable is accessed, or at which an exception is handled. Further join points in AspectJ 1.5 are constructor calls or execution, static initialization execution, object pre-initialization, object initialization, and advice execution (cf. AspectJ-Team, 2003).

In AspectJ, a *pointcut* is an explicit language construct used to select a set of join points based on different conditions. It can expose also context data of the selected join points. AspectJ provides various primitive *pointcut designators*, which a programmer can compose to define anonymous or named pointcuts. A pointcut designator can be thought of as a condition that is matched dynamically against the join points during execution. For example the pointcut designator *'call(void Database.get(Object))'* matches all method call join points that call the *get* method on an instance of the *Database* class. Another pointcut designator matches join points based on the class where it is defined, so *'within(ClientA)'* matches all join points inside the *ClientA* class or its instances, e.g., all method execution, method call, class instantiation, or constructor call join points. Individual pointcut designators can be combined with the operators *'||'*, *'&&'*, and *'!'* like expressions in Java. For example, *'call(void Database.get(Object)) && !within(ClientA)'* matches all method call join points to the specified method except those located in the *ClientA* class. This way, it is possible to create complex pointcuts from few primitive pointcut designators.

The definition of a *named pointcut* is similar to the definition of a member variable or a method in a class. The syntax is *'[modifier] pointcut [pointcutname]() : [pointcut designators];'*, so the above pointcut might be defined as *'public databaseaccess(): call(void Database.get(Object)) && !within(ClientA);'*. Named pointcuts can be used in other pointcuts, e.g., *'databaseaccess() && within(ClientB)'*.

As mentioned above, a pointcut can also expose data about the context, specifically the current and the target objects and method parameters if those are available at the matched join point. The values are then defined as parameters of the pointcut and collected with the pointcut designators *this*, *target*, or *args*. For example, to expose the parameter from the *get* method we extend the pointcut like this: *'public databaseaccess(Object key): call(void Database.get(Object)) && !within(ClientA) && args(key);'*.

In Table 2.1 we list all available pointcut designators in AspectJ. Most pointcuts are matched based on *lexically comparisons* and allow pattern expressions. For example the

[6] `http://www.eclipse.org/aspectj/`

call pointcut designator compares the specified method signature with the signature of the join point. Using pattern expressions it is also possible to match different signatures, e.g., *'call(* get*(..))'* matches all calls to methods whose name start with 'get'.

Although pointcuts to match join points dynamically in the program execution, for most pointcuts the set of matched join points can be evaluated statically (Kiczales et al., 2001). Pointcuts that match exactly one join point are called *heterogeneous*, while pointcuts which match more than one join point are called *homogeneous* (Apel et al., 2006c,a).

Advice in AspectJ is a method-like mechanism that describes the action that should be performed when a join point is matched with a pointcut. The advice statement can be executed *before*, *after* or *around* the join point. An advice statement has the following syntax: *'after() : [pointcut] { [advice body] };'*. An advice statement can declare parameters, however these must be bound with values exposed by the pointcut, for example *'after (Object key) : call(void Database.get(Object)) && args(key) { print(key); }'*.

The body of advice is similar to a method body. Only values declared in the advice declaration can be used. Additional information about the extended join point can be accessed through a special *thisJoinPoint* keyword. Another special case is *around* advice which implements code around the matched join point. It can either call the original join point with the *proceed* keyword, or replace the join point with a different implementation.

Static Crosscutting. An additional way to extend a program in AspectJ is through static crosscutting. Static crosscutting modifies the structure of a program statically, e.g., introduces new methods or alters the inheritance hierarchy. Static crosscutting is not part of every aspect-oriented language and not included in the AOP definition of quantification.

The introduction of methods or member variables is done with a language construct called *Inter-Type Member Declaration (ITMD)*. This allows developers to introduce a method or a member variable into other types. The syntax for an ITMD is similar to a method or member variable declaration, but has the name of the target type as a prefix to the name, e.g., *'public void Database.lock() { [implemenation] }'* or *'private int Database.lockCount=0;'*.

Static crosscutting in AspectJ furthermore can change the inheritance hierarchy. Thus, it is for example possible to introduce a new interface and let various existing classes implement this interface, e.g., *'declare parents: *Node implements Comparable;'*). With a combination of *declare parents* and an ITMD it is even possible to introduce members homogeneously into multiple types.

A static crosscutting construct usually used for policy enforcement allows to describe pointcuts that create a compiler error or warning whenever a join point is matched by this join point. As this must be calculated at compile time, only pointcuts that can be matched statically are possible. An example of this construct is *'declare error: call(void Database.get(Object)) && within(Database): "do not call the get method internally"'*, which prevents compilation with an error message when any join point exists in *Database* that calls the *get* method.

Pointcut Designator	Matched Join Points
call(*MethodPattern*)	Method call join point whose signature matches *MethodPattern*.
execution(*MethodPattern*)	Method execution join point whose signature matches *MethodPattern*.
get(*FieldPattern*)	Field reference join point whose signature matches *FieldPattern*.
set(*FieldPattern*)	Field set join point whose signature matches *FieldPattern*.
call(*ConstrPattern*)	Constructor call join point whose signature matches *ConstrPattern*.
execution(*ConstrPattern*)	Constructor execution join point whose signature matches *ConstrPattern*.
initialization(*ConstrPattern*)	Object initialization join point whose signature matches *ConstrPattern*.
preinitialization(*ConstrPattern*)	Object pre-initialization join point whose signature matches *ConstrPattern*.
staticinitialization(*TypePattern*)	Static initializer execution join point whose signature matches *TypePattern*.
handler(*TypePattern*)	Exception handler join point whose signature matches *TypePattern*.
adviceexecution()	All advice execution join points.
within(*TypePattern*)	All join points where the executing code is defined in a type matched by *TypePattern*.
withincode(*Pattern*)	All join points where the executing code is defined in a method or constructor whose signature matches *Pattern*.
cflow(*Pointcut*)	All join points in the control flow of any join point P picked out by *Pointcut*, including P itself.
cflowbelow(*Pointcut*)	All join points in the control flow of any join point P picked out by *Pointcut*, but not P itself.
this(*Type*)	All join points where the currently executing object is an instance of *Type*.
target(*Type*)	All join points where the target object is an instance of *Type*.
args(*Type*, ...)	All join points where the arguments are instances of the appropriate type.
if(*BooleanExpression*)	All join points where the boolean expression evaluates to true.

Table 2.1: AspectJ Pointcut Designators in AspectJ 1.5 (AspectJ-Team, 2003).

```
1  public aspect Logging {
2    static interface Traceable {
3      String getState();
4    }
5    declare parents: (Transaction || Database) implements Traceable;
6    public String Transaction.getState() { /*return state for logging.*/ }
7    public String Database.getState() { /*return state for logging.*/ }
8
9    private pointcut loggedMethods(Traceable obj): call(* Database.*(Traceable)) &&
         args(obj);
10   after(Traceable obj): loggedMethods(obj) {
11     trace(thisJoinPoint.getSignature.toString(), obj.getState());
12   }
13   private void trace(String signature, String state) { /*print trace message*/ }
14 }
```

Figure 2.7: AspectJ Aspect Example.

Aspects. To implement pointcuts, advice or static crosscuts, a developer declares aspect types. Aspect types are AspectJ's modules to implement crosscutting concerns. Aspect declarations are similar to class declarations: they define the type and its implementation. They are either defined as top-level aspects in own source files or as inner-aspects of classes. Aspects can contain pointcuts, advice, and the described static crosscuts which allow them to crosscut other types. Furthermore, aspects can contain all possible class members like methods or inner interfaces (Kiczales et al., 2001; AspectJ-Team, 2003).

A major difference between aspect and class types in AspectJ is that aspects are not instantiated directly with an *new* expression, but are instantiated automatically by the system. Aspects are initiated usually as singletons, but special instantiation commands based on pointcuts are possible (cf. AspectJ-Team, 2003).

As an example we show an aspect that implements the crosscutting concern logging and uses most language constructs introduced in this section in Figure 2.7. First, this aspect declares an inner interface (Lines 2–4), then it changes the inheritance hierarchy so that the *Transaction* and *Database* classes implement this interface (Line 5). ITMDs are used to implement the interface's method for both target classes (Lines 6, 7). Next, a pointcut is defined that matches all call join points to methods in the *Database* class which use an instance of the *Traceable* class as parameter (Line 9). Finally, an advice statement uses this pointcut to call a *trace* method of the aspect (Lines 10–12), which implements the tracing logic (Line 13). Thus, the aspect modularizes the crosscutting concern that otherwise would be scattered over various classes in an object-oriented implementation.

2.6 AHEAD

AHEAD (short for *Algebraic Hierarchical Equations for Application Design*) is a model of FOP based on *step-wise refinement*. An accompanying tool suite supports program development in an AHEAD fashion (Batory et al., 2004). Step-wise refinement is a paradigm that aims at building complex programs from simple programs by adding features incrementally.

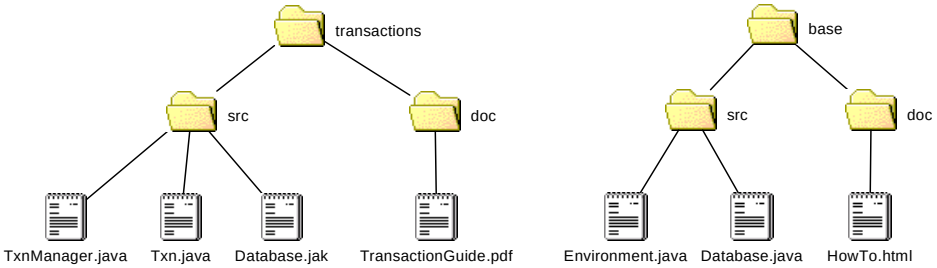


Figure 2.8: AHEAD Feature Modules.

In AHEAD these features are the primary abstractions of software modularity. Feature in AHEAD can encapsulate different program representations like code, make files, documentations, regression tests or aspects (Batory, 2004). A program is composed by sequentially applying features to a base code, i.e., features are program transformations (Batory et al., 2004).

In AHEAD each feature is implemented by a feature module in an distinct directory. This directory is a *"containment hierarchy of artifacts that can include multiple representations of an application"* (Batory et al., 2004). This means that a directory can contain different types of artifacts. Typical artifacts are source code files like Java files or aspects, HTML files that contain documentation, and make files. All these artifacts can be refined in a step-wise manner, i.e., as functions. It is possible to define functions that transform arbitrarily nested containment hierarchies, down to a primitive artifact. These functions can modify the containment hierarchy by either adding new nodes, e.g., adding new Java classes or HTML files, or by modifying existing ones (Batory et al., 2004).

In Figure 2.8 we show an example of two features *base* and *transactions*. Both features are defined in their own directories and have their own subdirectories to structure files. The feature *base* implements a database engine with the two files *Environment.java* and *Database.java* and also provides a documentation in form of an HTML file. The feature *transactions* contains the transformation that can be applied to the base feature. It introduces two new classes in *Txn.java* and *TxnManager.java* and a new documentation *TransactionGuide.pdf*. It also extends the *Database* class in a way that it uses the new transactions. This transformation is specified in a AHEAD-specific Jak format[7]. An application can be composed of either only the base code for a database engine without transaction support, or of both features for a transactional database engine. To compose features, all introductions are combined in a target directory and all transformations are applied. In our example when both features are composed the hierarchy depicted in Figure 2.9 is created, where *Database.java* is a modified version of the original *Database.java* file.

Most tools of the AHEAD Tool Suite deal with composing features and performing transformations for different code artifacts. For example, the *composer* tool is used to manage the composition of different features. Tools like *mixin, jampack, xak, balicomposer*, or

[7]Jak is a superset of Java, used to describe extensions to Java sources. The Jak language is outside the scope of this paper, for further information confer Batory et al. (2004).

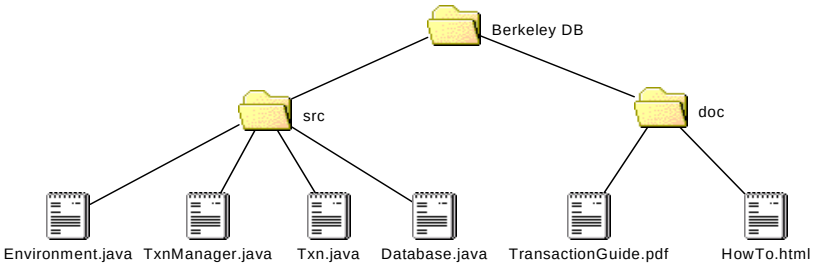


Figure 2.9: Composed AHEAD Feature Module.

bcmixin are used to compose specific artifacts like Jak files, XML files, grammar files, or Java byte code files.

The AHEAD Tool Suite also provides sophisticated tools to integrate the features just shown with a *feature model* as introduced in Section 2.3. For example, the *guidsl* grammar can be used to specify the feature model including its constraints. The *guidsl* tool can then be used to visualize this feature model, to specify a configuration, to check constraints and design rules, and to debug the model (Batory, 2006). Due to the integration of feature model and features, the visual selection of features can output a configuration file that is directly used as input for the composition tools. Thus, the constraints of the feature model can be enforced. The tool suite also provides an IDE which integrates the feature model to browse and query features and to compose applications.

2.7 Summary

In this Chapter we described the architecture and the concepts of Berkeley DB, which are required to understand the features refactored and discussed in this thesis.

Additionally, we introduced FOP and the concepts of features and feature models to describe a domain. A graphical notation for feature diagrams and a grammar to describe feature models were introduced. These are used later to describe features and their relations in Berkeley DB and to reason about the composition.

Afterwards, we presented the concepts of AOP and the aspect-oriented language AspectJ. In the remainder of this thesis we use AspectJ to implement features. To organize the source code of features, we use AHEAD, a model with an accompanying tool suite for feature-oriented programming.

Chapter 3

Refactoring Berkeley DB with AspectJ

In this chapter we present our results in refactoring Berkeley DB with AspectJ. We first evaluate the infrastructure necessary for refactoring, followed by a description of how features have been identified. Then the refactoring process is described, including some typical examples. Finally we conclude with the analysis of the refactored features and a feature model.

3.1 Infrastructure

Before the actual refactoring can be done it is necessary to provide an infrastructure for composing the application from features. Unfortunately, AspectJ does not provide any model or infrastructure how to handle optional aspects so that certain features can be removed from the application at compile time. Furthermore it provides no cohesion mechanism to group aspects to a feature (cf. Lopez-Herrejon et al., 2005). The straightforward approach is to encode a feature in one aspect and include or exclude this file from compilation.

While configuration by manipulating the build path seems reasonable, AspectJ does not provide a mechanism to handle dependencies or other higher level interactions. Additionally, encoding a feature completely in one aspect is possible as shown in Section 2.5, but often seems counterproductive as also observed by Murphy et al. (2001). We experienced that refactoring of features is very complex and requires to write many AOP language constructs like advice or pointcuts. Therefore it became necessary to split large and complex aspects into several pieces that are aspects as well that belong all to one feature to keep it maintainable.

Consequently, the demand for an infrastructure suitable for refactoring features arose, so that we decided to use the infrastructure of the FOP tool suite AHEAD (Batory et al., 2004). In AHEAD, features are stored in different directories, also called containment hierarchies, where each directory can contain multiple artifacts that belong to one feature. We followed this approach and created a distinct directory for every feature which each contain one or more aspects and optionally some classes.

Furthermore, AHEAD provides an infrastructure to create feature models with the ability to express relationships, constraints, etc. (Batory, 2005), that we could reuse in our project. To compile a project with a certain configuration AHEAD calls its own compiler with an equation file listing all included feature directories. Similarly, in our project the build path for the AspectJ compiler is adjusted for every configuration, so that the according directories are included and compiled with their aspects and classes. This way, because AspectJ does not have an own infrastructure, we impose the AHEAD infrastructure on this AspectJ project.

3.2 Identification of Features

One of the most important questions in refactoring is the identification and selection of features. Because the aim of this project is to refactor Berkeley DB into features and not to refactor technical crosscutting code into aspects, it is necessary to search for conceptual features instead of using Aspect Mining technologies to search for scattered code (cf. Section 2.4). In this section we show how we located features and which potential features we found by first reading the manual, then analyzing the parameters, and finally browsing the source code.

We started to search for features using Berkeley DB's 'Getting Started Guide' (Sleepycat, 2006, p. 1f). The manual proudly presents the 'major features' of Berkeley DB. We therefore searched for potential features in this list and tried to find and understand the according pieces of code to evaluate a potential refactoring. The following list sums up our findings for each 'major feature' described in the manual.

- **Large database support.** The support for large databases is a fundamental design concept and influenced design decisions on almost every part of the application, especially persistence and caching. It is not possible to refactor a large database support feature, because the developer could not decide which design decisions and which code would belong to such a feature.

- **Multiple thread and process support.** The synchronization code required for this multi-thread and multi-process support could be refactored in one or more features. The refactoring into optional features could even increase performance when multi-thread support is not required.

- **Database records.** The decision to support arbitrary objects as keys and values in the database can be refactored to a feature. This feature would contain all the code that is necessary to support non-trivial keys and large value objects and could potentially improve code readability and maintainability[1].

- **Transactions.** Similar to multi-threading, transaction management requires lots of code. Transactions in Berkeley DB are already optional and can be switched off at

[1] The remaining code could, for example, only support Java's primitive type `long` as keys and only values with a fixed maximum size that always fit into buffers.

runtime when the database is loaded. Refactoring transaction code into an optional feature might provide benefits in code size and understandability. Turning transactions off at build time might provide an increase of performance over the design time switches and dummy objects currently used by Berkeley DB (cf. Section 2.1).

- **Indexes.** The support for indexes[2] can be refactored into an optional feature for code size reduction and improved understandability.

- **In-memory cache.** The in-memory cache of Berkeley DB is closely linked to other parts of the caching and buffering subsystem as described in Section 2.1. The code might be decomposed into different features for improved maintainability. The cache or parts of it might also be replaced later by a different implementation in a different feature, allowing the user to select which caching features and which implementations he wants to use in his Berkeley DB configuration.

- **Log files.** As described in Section 2.1, the persistence layer of Berkeley DB uses no-rewrite log files and thereby differs from most other database systems. The refactoring of the whole persistence layer or parts of it might improve understandability and might even allow to exchange the no-rewrite log file approach with a different implementation of the persistence layer, for example one using a traditional database file.

- **Background threads.** The four background threads INCompressor, Evictor, Cleaner, and Checkpointer (cf. Sections 2.1, 2.1, and 2.1) represent broader design concepts in Berkeley DB. A refactoring seems only reasonable concerning their nature as background daemons. The actual tasks these threads perform belong to concepts of the B^+-tree, the persistence layer and caching. Nevertheless it might be useful to refactor the concept behind these threads into a certain feature, e.g., the lazy deletion of tree nodes using the INCompressor or the asynchronous writing of INs using the Checkpointer. This way these conceptual features are optional or even replaceable.

- **Database environments.** As described in Section 2.1 the environment object encapsulates multiple databases. This is a fundamental architectural decision on which the persistence system, the access methods, and the concurrency implementation rely. Additionally, the environment object is used all over the implementation for global settings and objects. A refactoring might break up these dependencies, but presumably it is very difficult because the environment design is a fundamental design decision.

- **Backup and restore.** The restore operation is an elemental part of the environment's loading process, required by the no-rewrite log persistence layer. Backups can be done by copying files without any further tool support for the same reason. Consequently backup and restore are currently not individual features. Nevertheless such

[2] Indexes are also called *secondary databases* in the Berkeley DB terminology (cf. Section 2.1).

features would be necessary if a different persistence layer implementation was chosen.

- **JCA and JMX.** Both the *Java Connector Architecture (JCA)* and the *Java Management Extensions (JMX)* are extensions build on top of Berkeley DB to integrate it with Java and especially *Java 2 Platform, Enterprise Edition (J2EE)* applications. These extensions could be easily refactored into different features, but were not considered in this project to remove dependencies on external libraries and concentrate on the core system of Berkeley DB.

Overall this list of 'major features' from the manual allows to derive many features. Although it is necessary to keep in mind that the previous analysis of features is already based on an additional study of the source code, the manual provided a good starting point for the understanding of Berkeley DB's architecture.

The features derived from the manual are usually very large, like the whole transaction or persistence layer, and they are fundamental design decisions, like the environment object or the no-rewrite log concept. Furthermore they mainly reflect those features that are special to Berkeley DB. For this case study a more balanced selection of small and large features is necessary.

An alternative starting point are the configuration parameters of Berkeley DB, because they sum up a list of variation points, often with the possibility to disable certain features at runtime. The following list shows some potential features derived by a configuration parameters analysis.

- **Memory settings.** The user of Berkeley DB can set various parameters concerning the memory allocation, like maximum overall amount of memory or amount of memory for caching and for buffering. So, it seems plausible to refactor the memory management and allocation to one or more features.

- **Read only operation.** The environment can be created read only. To improve performance and maintainability all operations using write access can be refactored into a separate optional feature, drastically reducing the code size of the database engine.

- **Transaction, locking and latches configuration.** It is possible to configure transactions, locking and thread synchronization with latches on a detailed level, e.g., the user can set a timeout for transactions, the size of lock tables, or the isolation level. These variation points appear to be a reasonable target for refactoring, enabling the user to configure the database engine at build time.

- **I/O buffers.** It is possible to select the individual size of memory and the amount of buffers used for I/O operations. The I/O buffers appear as reasonable feature for refactoring, and might be a target for alternative implementation.

- **I/O operations.** There are five implementations for I/O operations. The user can chose between different I/O frameworks and synchronized or chunked versions. This

appeares as a good opportunity for alternative features where only the code of the required features is compiled.

- **Checksum read.** Berkeley DB calculates a checksum for every entry it writes into the log and validates it during reading. Although these checks are fast and the checksums small, checksum writing and validation can be refactored into optional features to increase performance and decrease log file size. This way not only the reading but also the writing of checksums can be switched off.

- **Memory-only log.** It is possible to configure Berkeley DB to run only in memory, without writing a log file. Even though this does not scale for large amounts of data, it indicates that there are applications where the persistence layer is not required. Therefore refactoring the complete persistence layer to an optional feature seems desirable, to decrease code size and increase performance.

- **File handle cache size.** The file handle cache size parameter indicates the existence of a file handle cache and motivates further code inspection. The file handle cache is a small cache that can be refactored and is expected to improve code readability.

- **Evictor and Cleaner settings.** Various settings for the Evictor and the Cleaner (cf. Section 2.1 and 2.1) indicate that a refactoring of the these subsystems, by having all related design decisions in one feature might improve overall code readability. Some parameterizable design decisions like the Evictor's order or the Cleaner's internal look ahead cache might again be refactored into separate features.

- **Debug options.** Berkeley DB provides some debug options to log data, to check leaking locks, and to yield threads after 'strategic points'. These debug options are usually switched off and provide utilities only for very specific purposes. They seem like natural optional features, improving readability by freeing the database code from debug code.

This list illustrates that the configuration parameters are a good source for features as well. Contrasting the features described in the manual, this time we found features of varying size and also some general purpose features like logging and I/O operations.

The above two methods to find features often require to read the source code for a better understanding of the concept behind the described feature or the parameter. Nevertheless we found some potential features only by carefully studying the source code. Often the source code reading was initiated by another feature from one of the lists above. The following list shows some additional features that were found while reading the source code or refactoring other features[3].

[3]This list includes only conceptual features, not implementation features. We also identified various implementation features, but we omitted them from the list for a better overview of the more important conceptual features.

- **Statistics.** Berkeley DB collects statistics about various parameters all over the application. This includes statistics about almost every major part of the system, including the overall memory usage, the number of locks held, or the data to calculate the cache's buffer hit ratio. The code to collect these statistics is always executed independent of whether the statistics are required or not. Consequentially, it is reasonable to refactor them into an optional feature to increase code readability or even performance.

- **Database operations.** When analyzing the parameters, we found that refactoring all write operations into a feature would be possible. When reading the source code, we observed that it is possible to refactor the individual database operations, because each operation usually introduces its own code fragments. For example the implementation of the *remove database* operation is about 200 LOC and spreads across multiple classes, although it is only executed for this particular operation and reused only once for the *truncate database* operation. To refactor operations like *get*, *put*, *remove*, *truncate* or *create* into individual features might improve readability and maintainability, and makes dependencies between these operations explicit.

- **Disk full handling.** Another specific piece of code checks I/O exceptions for a possible full disk cause. In this case the implementation tries to perform some emergency operations and writes over existing code in the log file, as a clear exception to the general concept of the no-rewrite log. This code can be refactored into a feature because it injects some specific code for one specific use case in various classes. Thereby code readability can be improved.

- **Critical eviction.** Critical eviction is a part of the Evictor subsystem. The Evictor runs usually as background thread but before certain operations it is additionally called synchronously to ensure enough space is available for the operation. These synchronous calls are called critical eviction. Critical eviction is another conceptual feature that can be refactored.

This list shows that it is possible to find additional features by reading the source code.

Overall, we found lots of potential features and have a good selection features of varying size. We observed that searching for features starting on a high level using the manual and configuration parameters is efficient to find conceptual features, but on a lower level reading the source code even more features can be found. We estimate that further analysis will find more features. The refactoring of the identified features seems promising, because an improved readability, maintainability and performance is expected.

3.3 The Refactoring Process

The refactoring process followed a simple pattern: after the identification and selection of a feature, all source code that belongs to this feature is tagged and removed; afterwards it

is reintroduced again using AspectJ. Formally, it would be possible to use only the refactoring *Extract Feature into Aspect* as described by Monteiro (2004, p. 5ff) and Monteiro and Fernandes (2005), however this description is vague and informal, and tool support is not available. Hence we use a manual approach described in this section and illustrated by examples taken from our refactorings of the memory budget and statistics features.

To tag and remove a feature's source code we used an interactive and iterative approach. Starting with the code section found when identifying a feature, e.g., interface methods or the parameter definition code, we commented the code out, using a special tag inside the comment. By searching iteratively for code that requires the removed code or is called by the removed code, we could usually find most parts of a feature's code basis.

For example, by removing the *MemoryBudget* class of the memory budget feature, we were able to find various calls to this class by attempting to compile the code. Based on these code fragments, we identified member variables used only in this feature. Even though the variables do not create a compilation error when left inside the code, collecting data for the memory budget feature is their only purpose, so that we could tag and remove them as well. In the next iteration we found and removed all code that reference these variables, usually to update the current memory usage. This way, we found the whole code basis of the memory budget feature with only few iterations.

After we removed all code belonging to a feature, we tested the application for correct behavior without this feature as shown below, given that the feature was optional. In a final step we reintroduced the tagged code with AspectJ. Which AspectJ language constructs were used for this reintroduction to recreating the old behavior depends largely on the type of code refactored. In the following sections we show typical refactorings we used in this project.

Basically there are two different ways how code can be reintroduces using AspectJ, it can either introduce new classes or methods, or it can modify existing methods. Therefore we discuss static introductions and method refinements separately. Afterwards we show AspectJ's advanced concept of homogeneous extensions and our approach to access local variables. Finally we conclude by explaining how we verified correctness.

3.3.1 Static introductions

The simplest refactoring is to move whole interfaces, classes, methods or member variables to a feature, where they are reintroduced statically. In contrast to method refinements, it is not necessary to extend or modify existing methods, but also not possible to implement interactions.

The refactoring of a class is simple and can be done without AspectJ constructs because of the AHEAD infrastructure we imposed on the project (cf. Section 3.1). Consequentially, a class is simply moved to the feature directory, where it is only included in the build path if the feature is selected. In case that the package is changed when moving the class, the object-oriented *Move Class* refactoring (cf. Pizka, 2004) is applied. Finally, it is also possible to move the class as an public inner class into an aspect as described by Monteiro and Fernandes (2005) as the *Inline Class within Aspect* refactoring. Interfaces are treated

equally.

Method and member variable refactorings can be achieved by AspectJ's ITMDs. The code is removed from the class and reintroduced using an ITMD inside an aspect (cf. Laddad, 2003, p. 95f). This refactoring is also known by the names *Move Method from Class to Inter-type* respectively *Move Field from Class to Inter-type* (Monteiro and Fernandes, 2005) or *Extract Introduction* (Hanenberg et al., 2003). Figure 3.1 shows an example of a refactoring with ITMDs.

```
1  public class IN {
2
3    private long inMemorySize;
4
5    void initMemorySize() {
6      inMemorySize = computeMemorySize();
7    }
8
9    long computeMemorySize() {
10     MemoryBudget mb = databaseImpl.
           getDbEnvironment().
           getMemoryBudget();
11     ...
12     return calcMemorySize;
13   }
14
15   ...
16 }
17
18 public class EnvironmentImpl {
19   ..
20   public MemoryBudget getMemoryBudget(){
21     ...
22   }
23   ...
24 }
```

(a) Original code.

```
1  public class IN {
2    ...
3  }
4
5  public class EnvironmentImpl {
6    ...
7  }
8
9  public aspect BudgetAsp {
10   private long IN.inMemorySize;
11
12   void IN.initMemorySize() {
13     inMemorySize = computeMemorySize();
14   }
15
16   long IN.computeMemorySize() {
17     MemoryBudget mb = databaseImpl.
           getDbEnvironment().
           getMemoryBudget();
18     ...
19     return calcMemorySize;
20   }
21
22   public MemoryBudget EnvironmentImpl
         .getMemoryBudget(){
23     ...
24   }
25 }
```

(b) Refactored code.

Figure 3.1: Static introductions using AspectJ's ITMDs.

Monteiro and Fernandes (2005) consider the static introduction of code using ITMDs as a 'bad smell' called *Aspect Laziness* if the additional code could be shared between instances or is used only inside the feature. They propose to use aspect methods and a map instead of ITMDs. For example, the refactoring of the feature STATISTICS introduced many variables for temporary counters used only internally in the feature. The change from ITMDs to aspect methods and maps can be achieved by applying the refactorings *Replace Inter-type Method with Aspect Method* and *Replace Inter-type Field with Aspect Map* described by Monteiro (2004, p. 28ff). It can alternatively also be achieved using the *perthis* association of the aspect (cf. Laddad, 2003, p. 122ff) and member variables. The latter approach is easier to implement because references to to original objects are not needed in advice statements. On the other hand it is criticized for its weak conceptual structure (Mezini and Ostermann, 2004), it increases overhead costs, and the method or

variable can no longer be accessed from outside the aspect. Figure 3.2 compares all three approaches on an example from the statistics feature.

Altogether we reintroduced 4 interfaces, 58 classes, 365 methods and 213 member variables during our refactoring of Berkeley DB. We only used the *perthis* association approach in 6 cases and ITMDs in all others, because ITMDs are straightforward and the methods introduced are often called by other features.

3.3.2 Method refinements

Static introductions can only introduce code but cannot implement interactions. Therefore, it is also necessary to extend existing methods. These extensions are called method refinements. In AspectJ method refinements are implemented using advice (cf. Section 2.5).

Even though many papers simply name the refactorings of these extensions *Extract Advice* or *Extract Fragment into Advice* and do not distinguish between different cases (e.g., Monteiro and Fernandes, 2005, 2004; Iwamoto and Zhao, 2003; Hanenberg et al., 2003), there are various different refactorings that can be applied to move code from a method to an advice declaration. The selected refactoring depends on the structure and position of the target code that should be refactored inside the method.

Extract Simple Method Refinement. The easiest refactoring for a method refinement, which we call Extract Simple Method Refinement (ESMR), is used when the target code is placed at the beginning and/or the end of a method. It can also be used if it is possible to move the target code to the beginning or the end of the method without behavior changes.

In these cases the method's execution join point can be extended with a *before* or *after* advice statement. The target code is then moved from the method to the advice. If the target code frames the original method, e.g., with a try-finally statement, *around* advice can be used. Figure 3.3 shows an example taken from refactoring the memory budget feature, reintroducing a method call at the beginning of a method.

We used the ESMR refactoring 214 times in the Berkeley DB project. This refactoring is simple and sound. It is also described as *Extract Beginning* respectively *Extract End* (Binkley et al., 2005, 2006), or *Extract Advice* refactoring (Hanenberg et al., 2003) and proven formally by Cole and Borba (2005).

Extract Call Refinement. In cases where the target code is not at the beginning or the end of the method and cannot be moved there, other refactorings are needed. If the target code is placed before or after a certain method call in the code, often a refactoring we call Extract Call Refinement (ECR) can be used. Using this refactoring, the call join point of a method call next to the target code is advised. Figure 3.4 shows an example of an ECR refactoring, where the target code is located directly next to the *adjustCursorsForInsert* method.

Because of limitations in the pointcut language of AspectJ this join point cannot be quantified directly, but must be narrowed down with a combination of a *call* and a *withincode* pointcut. We therefore also use the term *statement extension emulation*, because

```
1  class LogBufferPool {
2    private long nCacheMiss = 0;
3    LogBuffer getReadBuffer(long lsn) throws DatabaseException {
4      ...
5      if (foundBuffer == null)
6        nCacheMiss++;
7      return foundBuffer;
8    }
9    ...
10 }
```

(a) Original code.

```
1  public aspect LogBPoolStats {
2    private long LogBufferPool.nCacheMiss = 0;
3    after(LogBufferPool lbp) returning (LogBuffer foundBuffer) :
4      execution(LogBuffer LogBufferPool.getReadBuffer(long)) && this(lbp) {
5      if (foundBuffer == null)
6        lbp.nCacheMiss++;
7    }
8    ...
9  }
```

(b) Refactored code using ITMDs.

```
1  public aspect LogBPoolStats {
2    private WeakHashMap pool2nCacheMiss = new WeakHashMap();
3    after(LogBufferPool lbp) returning (LogBuffer foundBuffer) :
4      execution(LogBuffer LogBufferPool.getReadBuffer(long)) && this(lbp) {
5      if (foundBuffer == null) {
6        Long nCacheMiss = (Long) pool2nCacheMiss.get(lbp);
7        if (nCacheMiss == null) nCacheMiss = new Long(0);
8        pool2nCacheMiss.put(lbp, new Long(nCacheMiss.longValue() + 1));
9      }
10   }
11   ...
12 }
```

(c) Refactored code using aspect map.

```
1  public aspect LogBPoolStats perthis(execution(* LogBufferPool.*(..))) {
2    private long nCacheMiss = 0;
3    after() returning (LogBuffer foundBuffer) :
4      execution(LogBuffer LogBufferPool.getReadBuffer(long)) {
5      if (foundBuffer == null)
6        nCacheMiss++;
7    }
8    ...
9  }
```

(d) Refactored code using perthis association.

Figure 3.2: Different Versions of Refactored Introductions.

```
1  public class IN {
2    ...
3    void rebuildINList(INList inList) throws
         DatabaseException {
4      initMemorySize();
5      inList.add(this);
6      for (int i=0; i<nEntries; i++) {
7        Node n = getTarget(i);
8        if (n != null) {
9          n.rebuildINList(inList);
10       }
11     }
12   }
13 }
```

(a) Original code.

```
1  public aspect BudgetAsp {
2    before(IN in) :
3      execution(void
           IN.rebuildINList(INList))) &&
           this(in) {
4      in.initMemorySize();
5    }
6  }
```

(b) Refactored code.

Figure 3.3: Extract Simple Method Refinement Refactoring.

we emulate statement join points—that are not supported by AspectJ—inside a method by advising call statements inside this method. Due to this approximation the refactoring is constrained to cases where the method contains only one call to the advised method, or alternatively to those cases when all calls inside a method can be advised with the same code.

In refactoring Berkeley DB, ECR refactorings were only used on the condition that the target code belongs semantically to the advised method call, and therefore really is an extension to the call. For example, in Figure 3.4 the memory usage must be updated every time the cursor is adjusted inside the *insertEntry1* method, therefore refactoring as a call extension is reasonable. A semantic connection between the target code and the advised method as condition is reasonable because otherwise further refactoring can move the advised call to another potentially optional feature, thus changing the join point or even creating implementation dependencies (cf. Section 3.5 and 5 for observations and discussions about dependencies). ECR refactorings are therefore considered fragile, if not semantically connected to the advised method.

We used this refactoring 121 times in the Berkeley DB project. The ECR refactoring is also described similarly by Binkley et al. (2005, 2006) as *Extract Before Call* respectively *Extract After Call* refactoring.

Extract Inner Method Refinement. In case the target code is inside a method, yet not before, after or around a single method call, but before, after or around a whole sequence of statements, e.g., an inner code block of a while loop, a different refactoring can be chosen, we call Extract Inner Method Refinement (EIMR). In this case the sequence of statements can be regarded as an 'inner method', i.e., a method nested inside another method as supported by languages like JavaScript, Ruby, or Pascal. Java does not support inner methods, instead the object-oriented refactoring *Extract Method* (cf. Fowler, 1999, p. 110ff) can be used to create a new method that is not nested but has the semantics of an inner method. After this refactoring the new method can be advised from an aspect using the ESMR or

```
1  public class IN {
2    public int insertEntry1(CR entry) throws DatabaseException {
3      ...
4      if (nEntries < entryTargets.length){
5        byte[] key = entry.getKey();
6        int index = findEntry(key, true, false);
7        ...
8        entryStates[index] = entry.getState();
9        nEntries++;
10       updateMemorySize(0, getEntryInMemorySize(index));
11       adjustCursorsForInsert(index);
12       setDirty(true);
13       return (index | INSERT_SUCCESS);
14     }
15     ...
16   }
```

(a) Original code.

```
1  public aspect BudgetAsp {
2    before(IN in, int index):
3      call(void IN.adjustCursorsForInsert(int)) && this(in) && args(index) && withincode(int
           IN.insertEntry1(CR)) {
4      in.updateMemorySize(0, in.getEntryInMemorySize(index));
5    }
6  }
```

(b) Refactored code.

Figure 3.4: Extract Call Refinement Refactoring.

ECR refactoring.

Figure 3.5 shows an example of an EIMR refactoring, taken from the memory budget feature. The first step is the object-oriented refactoring that extracts the statements between the target code into a new method, which is simply called *extractedMethod* in this example. After this refactoring the extracted method provides a join point for an ESMR refactoring, that moves the target code to an execution advice inside of the aspect.

Similar to the ECR refactoring, we use this refactoring only if the Extract Method refactoring is reasonable semantically, i.e., if a developer can justify the refactoring even without the subsequent ESMR or ECR refactoring. Additionally, we required that the subsequent refactoring is semantically connected to the extracted method, too. Equally, as the semantic condition for the ECR refactoring we require these semantic connections for stability reasons during ongoing refactoring and for code understandability.

We used the EIMR refactoring 15 times in the Berkeley DB project. It can be formally described by a combination of the Extract Method refactoring and the ESMR or ECR refactoring, thus there is no need for further proof.

Hook methods. In all other cases where neither the ESMR, the ECR, nor the EIMR refactoring can be applied, the *hook method refactoring* can be used as a last resort. In this refactoring an explicit join point is created for the target code, by creating an empty method and calling it from a position next to the target code. We call this empty method *hook*

```
1  public class IN
2    public boolean deleteEntry(int index, boolean maybeValidate)
3        throws DatabaseException {
4      ...
5      if (index < nEntries) {
6        updateMemorySize(getEntryInMemorySize(index), 0);
7        int oldSize = computeLsnOverhead();
8        for (int i = index; i < nEntries - 1; i++) {
9          setEntryInternal(i + 1, i);
10       }
11       clearEntry(nEntries - 1);
12       updateMemorySize(oldSize, computeLsnOverhead());
13       nEntries--;
14       setDirty(true);
15       ..
16     } ...
17   }
18 }
```

(a) Original code.

```
1  public class IN
2    public boolean deleteEntry(int index, boolean maybeValidate)
3        throws DatabaseException {
4      ...
5      if (index < nEntries) {
6        extractedMethod(index);
7        nEntries--;
8        setDirty(true);
9        ..
10     } ...
11   }
12   private void extractedMethod(int index) {
13     for (int i = index; i < nEntries - 1; i++) {
14       setEntryInternal(i + 1, i);
15     }
16     clearEntry(nEntries - 1);
17   }
18 }
19 public aspect BudgetAsp {
20   void around(IN in, int index) :
21     execution(void IN.extractedMethod(..))
22     && this(in) && args(index) {
23
24     in.updateMemorySize(in.getEntryInMemorySize(index), 0);
25     int oldSize = in.computeLsnOverhead();
26     proceed(in);
27     in.updateMemorySize(oldSize, computeLsnOverhead());
28   }
29 }
```

(b) Refactored code.

Figure 3.5: Extract Inner Method Refinement Refactoring.

method because its only purpose is to provide a join point—a hook—for further extension[4]. After the creation of the hook method, the ESMR or ECR refactoring can be applied.

The hook method refactoring cannot handle target code that frames statements, e.g., synchronization code or a surrounding try-finally block. In some cases such framing can be resolved with two hook method refactorings, one for the opening code and one for the closing code. In cases where the code cannot be split it is necessary to use an extract method refactoring without the semantic connection described above. As this refactoring is only done to create join points without semantic justification, we count these refactorings as hook method refactorings as well, opposed to the catalog specified by Monteiro and Fernandes (2005).

We consider hook methods as bad design, because they are empty or arbitrarily chunked methods in the source code, that are also present even if the refactored feature is not included. Hook methods do not serve any function except providing join points for aspects. Murphy et al. (2001) call them *"'unnatural' methods [...] with no apparent purpose"*. They therefore decrease the readability and maintainability of the source code. Consequently we tried to avoid hook methods and used the other refactorings wherever possible during our refactorings. Nevertheless we still had to use the hook method refactoring 164 times.

The hook method refactoring is also informally described as a "Refactoring of a field access or method call relationship" by Hannemann et al. (2003).

3.3.3 Homogeneous extensions

All previous refactorings focused on moving a single piece of target code to an aspect inside a feature. In cases where the same target code appears in multiple classes or multiple methods, AspectJ's ability for homogeneous extensions can be used, for both, static introductions and method refinements.

To introduce the same method or member variable into multiple classes a homogeneous ITMD can be used (cf. Section 2.5). However in our refactoring of Berkeley DB we never had to introduce the same method or the same member variable in multiple classes.

Also all method refinements shown in Section 3.3.2 can be done homogeneously using pattern expressions and the *or* operator in AspectJ's pointcut language (cf. AspectJ-Team, 2003; Laddad, 2003, p. 67ff). A pattern can be defined to extend multiple methods with a same advice created by an ESMR refactoring, or a pattern can be used to advise calls inside different methods with the same code gained from a ECR refactoring. When using hook methods, it is even possible to call the same hook method multiple times and then advise all calls homogeneously.

Formally this can be described as multiple individual refactorings where the duplicated advice is afterwards combined to a homogeneous one using the *Extract Pointcut* refactoring described and proven by Cole and Borba (2005).

[4]Hook methods are also known as *dummy methods* (de Alwis et al., 2000; Eaddy and Aho, 2006), though we prefer the term hook method because it indicates the usage of the method as hook for pointcuts.

3.3.4 Accessing local variables

A specific problem shared by all method refinements is the access to local variables from the aspect, also observed by Sullivan et al. (2005). AspectJ allows only to access parameters and member variables, but does not provide a concept to expose the context to access the local variables of a join point (Hanenberg et al., 2003). For example, in Figure 3.6 it is not possible to extract Lines 5–7, because they could not access the local variable *errors* from an advice statement. This section shows the workarounds we used in our refactorings.

```
1  public class Environment {
2    public String close() {
3      StringBuffer errors = new StringBuffer();
4      /* long code to check for errors */
5      if (errors.length >0) {
6        Tracer.traceError(Tracer.ENVCLOSE, errors);
7      }
8      environmentImpl.close();
9      return errors.toString();
10   }
11   ...
12 }
```

Figure 3.6: Required Access to a Local Variable.

Basically there are three different approaches to access a local variable, the first moves them to the enclosing class as member variables, the second uses a method object, and the third creates a hook method.

The first approach is simple: instead of a local variable a member variable of the enclosing class is used. Unfortunately this breaks common coding conventions and it is only behavior conserving under certain conditions. For the developer it becomes harder to see if a member variable is actually a local variable, that has only been defined as member variable for potential extension, or if it is a real member variable that stores object data and can be used in multiple methods. They are similar to global variables that are avoided in common programming practice. This coding convention break decreases readability and hence the overall code quality. Additionally, the member variable approach does only work when it can be guaranteed that the method is only executed by one thread at a time, i.e., in synchronized code or single thread applications. Otherwise the same variable can be used by two different executions of the same method causing incorrect behavior. Because Berkeley DB is an application that supports high concurrency, we did not use the member variable approach in our refactorings.

The second approach uses the object-oriented refactoring *Replace Method with Method Object* as described in Fowler (1999, p. 135ff). This creates an object for the method and moves local variables to members variables of that object. Contrasting the previous approach each method execution is encapsulated in an own object and therefore the synchronization problems do not occur. However there is no tool support for this refactoring in Eclipse and we consider method objects as counterintuitive and hard to read, therefore we have not used them in our refactorings.

Finally, the third approach to access local variables uses hook methods. The local variables that the target code requires are passed as parameters. This way AspectJ can advise the hook method and access the local variables by intercepting the parameters. To change a single local variable by the target code, it is returned by the hook method. Figure 3.7 shows an example from the ATOMICTRANSACTIONS feature, where an advice changes a local variable and needs to access another one. The hook method only returns the parameter directly while the aspect can change the return value with an around advice statement. When multiple local variables are changed, there is a problem because a Java method can only return one value. This can be solved either by an object that encapsulates multiple values or by using multiple hook methods instead.

```
1  public class LN {
2    ...
3    public long delete(DatabaseImpl database, ..., Locker locker) throws DatabaseException {
4      ...
5      LogEntryType entryType;
6      if (locker.isTransactional()) {
7        entryType = LogEntryType.LOG_DEL_DUPLN_TRANSACTIONAL;
8      } else {
9        entryType = LogEntryType.LOG_DEL_DUPLN;
10     }
11     ...
12     DeletedDupLNLogEntry logEntry = new DeletedDupLNLogEntry(entryType, ...);
13     ...
14   }
15 }
```

(a) Original code.

```
1  public class LN {
2    ...
3    public long delete(DatabaseImpl database, ..., Locker locker) throws DatabaseException {
4      ...
5      LogEntryType entryType = LogEntryType.LOG_DEL_DUPLN;
6      entryType = hook_getEntryType(locker, entryType);
7      ...
8      DeletedDupLNLogEntry logEntry = new DeletedDupLNLogEntry(entryType, ...);
9      ...
10   }
11   private LogEntryType hook_getEntryType(Locker locker, LogEntryType entryType) {
12     return entryType;
13   }
14 }
15 public aspect TransactionalLN {
16   LogEntryType around(Locker locker) :
17       call(LogEntryType LN.hook_getEntryType(Locker, LogEntryType))
18       && args(locker, LogEntryType) {
19     if (locker.isTransactional())
20       return LogEntryType.LOG_DEL_DUPLN_TRANSACTIONAL;
21     else
22       return proceed(locker);
23   }
24 }
```

(b) Refactored code.

Figure 3.7: Hook Method to Access and Change Local Variables.

Using hook methods to access local variables leads to the same problems as using hook methods to refine methods: the hook method does not have a semantic justification, but complicates the source code for solely technical reasons. Nevertheless in refactoring Berkeley DB we were frequently forced to used hook methods to access local variables.

The topic of accessing local variables with AspectJ is discussed detail in Section 4.2.4.

3.3.5 Verifying Correctness

The verification of the correctness of a refactored feature is an important and difficult task. The application must be correct with and without the feature.

The easiest way to demonstrate the correctness of the refactoring is to use proved refactorings only and rely on a tool to avoid errors. Unfortunately, there are only two tools available as a prototype that supports only some refactorings and could therefore not be used productively for our refactorings[5]. Furthermore this approach ensures correctness when the refactored feature is included in the application, but cannot ensure correctness of the application without the feature.

A common approach to verify an application is the use of a test suite designed for this application. Berkeley DB has a large test suite covering many parts of the application, including many of the refactored features. Theoretically the test suite can be used to test the composed application. Unfortunately this requires to observe the test suite during refactoring and to propagate changes like the renaming of a method to the test suite as well. Furthermore, when runtime decisions have been changed to build time decisions, it is also necessary to update the test suite. Additionally, the test suite supports tests only on the complete application where all features are included. To be able to test the application only with certain features, or even without any features, it is necessary to have an feature-oriented test suite, where each test is assigned to a feature and special tests are designed for feature interactions. During refactoring, this requires severe changes to the test suite, which again are hard to verify themself and it makes the whole refactoring process more complicated[6].

We therefore decided not to use the test suite, but to use only compiler syntax checks and simple runtime tests, for a basic verification. We are aware that these tests are not sufficient for a productive use of the refactored Berkeley DB system. However, other problems discussed later prevent such a productive use anyway, therefore our tests are sufficient to analyze the basic concept of refactoring Berkeley DB using AspectJ.

The problems of correct aspect weaving with AspectJ and fragile pointcuts are discussed in detail in Section 7.

[5]The tool *AspectJRB*, version 0.10 (Oct. 2003) is available at `http://dawis.icb.uni-due.de/?id=210`.

[6]On the other hand, for new application where the developer knowns why he creates tests and to which features they belong, feature-oriented tests suites were already used sucessfully (e.g., Hunleth et al., 2001)

3.4 Refactored Features

Using the approach presented in the previous section, we refactored 38 features. We selected the features based on the lists identified in Section 3.2. In the following list we give a short overview of the refactored features:

- **Logging.** The first feature we refactored was the LOGGING subsystem, which contains a *Tracer* class and various *trace* calls. Afterwards this feature was split into 10 different smaller features for different logging handlers[7] and logging levels.

- **Concurrency and transactions.** From the concurrency and transactions subsystem we refactored only some parts: the ATOMICTRANSACTIONS subsystem responsible for atomicity, the FSYNC feature for file synchronization, and the LEAKCHECKING that checks if all locks and transactions are released when the database is shut down. We also tagged and partly refactored the LATCHES feature that implements fine grained thread synchronization in many parts of the database, and we tagged the LOCKING feature but did not refactor it.

- **Statistics.** We refactored the STATISTICS system into a features. This feature collects various statistics from different parts all throughout Berkeley DB and introduces methods to query these statistics.

- **B^+-tree.** From the B^+-tree subsystem, we refactored various features. The EVICTOR feature encapsulates the basic code to free memory from the cache. It is called by the EVICTORDAEMON feature implementing its nature as a background thread and the CRITICALEVICTION feature implementing synchronous Evictor calls before certain operations. The INCOMPRESSOR and DBVERIFIER features remove deleted nodes from the tree and verify a valid status. Finally the TREEVISITOR feature provides a service class to traverse the B^+-tree for the STATISTICS and DBVERIFIER features.

- **Memory Budget.** We refactored the MEMORYBUDGET feature, which assigns memory to the individual caches and buffers, and observes the memory usage of Berkeley DB to call the Evictor when memory runs out.

- **Database operations.** To represent the possible refactoring of all database operations, we refactored the two operations to delete and truncate a database into the features DELETEDBOPERATION and TRUNCATEDBOPERATION.

- **Persistence layer.** All other features belong to the persistence layer. We refactored the I/O operations into five alternative features IO, NIO, CHUNKEDNIO, SYNCHRONIZEDIO, and DIRECTNIO for different I/O implementations. The Checkpointer

[7] Following the naming conventions from the Java Logging API, handlers are objects used for different output media. Berkeley DB implements three handlers to print logging messages to the console (LOGGINGCONSOLEHANDLER), to a file (LOGGINGFILEHANDLER) or to the database log (LOGGINGDBLOGHANDLER).

has a feature for its implementation as a daemon thread (CHECKPOINTERDAEMON) and two optional configuration parameters CPBYTESCONFIG and CPTIMECONFIG. The Cleaner is also refactored into a daemon thread CLEANERDAEMON and its look ahead cache feature LOOKAHEADCACHE. Finally we refactored the small persistence features checksum validation of all read and write operations (CHECKSUMVALIDATION), file handle caching (FILEHANDLECACHE), handling of out of disk errors (DISKFULLERRORHANDLING), and locking of the database directory (ENVIRONMENTLOCK).

The 38 refactored features are representative of the much longer list of possible features identified in Section 3.2 and provide a selection of features with different sizes. We stopped the refactoring when it became a routine—but still laborious—process, and we did not expect any further insights from more refactoring.

3.4.1 Feature size

Already during the identification of features in Berkeley DB, we noticed a significant difference in size between the features. This was confirmed by the refactoring process. To analyze the influence of the size on the refactoring process and other results discussed later, it is necessary to categorize the features their size.

To determine the size of a feature, we need metrics. A classification based only on the commonly used LOC metric is not suited because it does not credit the complexity of feature interactions. Features with with various small, collaborating extensions can be very hard to develop and maintain whereas features that are large in terms of LOC but only implement few long extensions can be simple. Therefore we use the number of introduced aspects and classes, the number of ITMDs and pieces of advice, and finally the number of affected types as additional metrics (cf. Table 3.1).

To collect these statistics we build two tools called *abcstats* and *ajdtstats*. The first is build upon the alternative AspectJ compiler *abc* (cf. Avgustinov et al., 2005). This tool uses the abstract syntax tree of the compiler to count various static source code statistics used in this paper. The second tool is build upon the *AspectJ Development Tools (AJDT)* Eclipse plug-in (cf. Clement et al., 2003) and collects dynamic statistics about the weaved aspects. Table 3.2 shows the size we measured for the refactored features.

Based on these metrics, the features can be roughly divided by size into four groups: small, medium, large and very large. Small features contain only one aspect and few pointcuts. For example the I/O features, LEAKCHECKING, ENVIRONMENTLOCK and FILEHANDLECACHE belong to this group. Medium features typically consist of few aspects, the number of advice and introductions is between 10 and 40. In contrast to small features medium features typically affect multiple classes. In these group there are eight features, among them EVICTORDAEMON, TREEVISITOR, and DELETEDBOPERATION. Large features contain many—in our features 70 to 350—advice declarations or ITMDs, usually distributed in more than 10 aspects. In this groups, there are five features: ATOMICTRANS-

Metric	Definition
LOC	Lines of code, excluding comments, empty lines, and lines three or less non-blank characters.
Aspects (Asp.)	Number of aspects that the feature contains, including both, top level and inner aspects.
Classes (Cla.)	Number of classes that the feature introduces, including both, top level and inner classes.
Advice (Adv.)	Number of advice declaration the feature contains, e.g., before, after, after returning, around.
ITMD	Number of methods or member variables introduced by the feature with Inter-Type Member Declarations.
Affected Types (AT)	Number of types (classes, interfaces, aspects) that are extended by the feature either with advice or with ITMDs.

Table 3.1: Used Metrics

ACTIONS, LATCHES, MEMORYBUDGET, STATISTICS, and LOGGING[8]. Very large features are features with more than 500 advice declarations or ITMD. Such very large features were not refactored as explained in the next section.

Overall, we refactored several features of very different size, thus we are able to analyze effects of scale in this project.

3.4.2 Very large features

The above selection of refactored features does not include very large features that were proposed as possible features in Section 3.2, like the whole transaction system, the whole persistence layer, or all write operations. This is due to the observed effect that refactorings were already hard to do and fragile in small or medium-sized features (cf. discussion in Section 7). The attempt to refactor the transaction system was canceled because of the enormous effort required and due to the fact that some problems, that already occurred in smaller features like the Parameter Introduction and Passing Problem in the ATOMIC-TRANSACTIONS and MEMORYBUDGET features, but that occurred much more frequently in these very large features.

Because the attempt to refactor the transaction system was stopped, we pursued another way to validate our assumption that refactoring would be possible and that the small scale problems also occur in very large features. We did so by tagging all source code that belonged to the transaction system. This way, it was easy to remove the code while still having markers to refactor it later once the basic problems are solved or an automatic tool is available. The tagging of the transaction system was done using a separate branch of the version control system *CVS* where all code for the transaction system was commented out.

[8]Even though the LOGGING feature is divided into 10 features, which are all small or medium sized each, the whole logging systems can be considered as a large feature.

No.	Feature	LOC	Asp.	Cla.	Adv.	ITMD	AT
1–10	LOGGING	1115	45	1	95	36	24
11	ATOMICTRANSACTIONS	715	13	1	34	49	19
12	LEAKCHECKING	43	1	0	1	3	2
13	FSYNC	130	1	2	1	2	1
14	LATCHES	1835	19	6	104	51	28
15	STATISTICS	1867	28	7	83	262	30
16	EVICTOR	371	1	3	4	13	9
17	EVICTORDAEMON	71	1	0	5	4	3
18	CRITICALEVICTION	63	3	0	7	3	7
19	INCOMPRESSOR	425	2	0	10	11	4
20	TREEVISITOR	138	1	1	7	16	9
21	DBVERIFIER	391	4	2	3	11	10
22	MEMORYBUDGET	958	14	1	64	53	28
23	DELETEDBOPERATION	226	2	1	13	17	13
24	TRUNCATEDBOPERATION	131	1	0	0	5	3
25	CHECKSUMVALIDATION	324	1	1	17	14	8
26	CHECKPOINTERDAEMON	110	2	1	7	6	4
27	CPBYTESCONFIG	41	1	0	3	4	5
28	CPTIMECONFIG	59	1	0	2	4	5
29	CLEANERDAEMON	129	1	0	6	3	2
30	LOOKAHEADCACHE	84	1	2	3	1	2
31	FILEHANDLECACHE	101	1	1	4	1	2
32	DISKFULLERRORHANDLING	41	1	0	1	3	2
33	ENVIRONMENTLOCK	61	1	0	3	2	2
34	IO	38	1	1	1	0	1
35	NIO	26	1	1	1	0	1
36	CHUNKEDNIO	52	1	1	1	0	1
37	SYNCHRONIZEDIO	26	1	1	1	0	1
38	DIRECTNIO	6	1	0	1	0	1

Table 3.2: Feature Size.

Using the *diff* tool it is now possible to find all the places where code must be reintroduced again with an aspect. Using this technique it is possible to estimate the required effort for the refactoring and to search and measure the transaction system's code.

After all, the refactoring of very large features is possible but problems already observed in smaller features make the refactoring of these very large features very time-consuming while not promising new insights. The experiences from the attempted refactoring and the marking of all transaction code support this assumption.

3.5 Feature Model

The previous section presented the refactored features independently. However, a simple feature list is not sufficient to express optional or alternative features or a Software Product Line. For the flexible usage and save composition it is necessary to describe the dependencies and constraints. As described in Section 3.1 we use a classic feature model and impose it on our AspectJ project.

Section 2.3 has shown that a feature model captures all these dependencies and allows to reason about the composition of the application. The feature model can be expressed in feature diagrams or grammars. A feature diagram for all 38 features is shown in Figure 3.8[9]. Furthermore a grammar representation using the *guidsl* format (Batory, 2005), is shown in Figures 3.9.

The first glance on the feature diagram shows that many dependencies exists and the overall feature model is quite complex, even though Berkeley DB in still a small Software Product Line, according to the number of refactored features. To discuss the whole feature model would exceed the scope of this thesis, therefore we only discuss some characteristics and analyze the basic types of dependencies.

Our first observation is that almost all features are optional. The high amount of optional features is caused by our approach to identify features. We explicitly searched the manual and the applications parameters to search for subsystems and design decisions that could be removed or replaced (cf. Section 3.2). Small required parts of the application, that implement a design decision and could be refactored into features, were not found with this approach. On the other hand large required conceptual features were found, but were usually part of fundamental design decisions like the B^+-tree or the non-rewrite log. These design decisions would result in very large features that were not refactored as explained in Section 3.4.2.

A second observation is that alternatives (cf. Section 2.3) are hardly used in the feature model. In most cases the user can configure Berkeley DB by selecting one or more subfeatures, in some cases even on a very detailed level like the logging subsystem shown in Figure 3.10, where the user can configure log levels and log handlers. In contrast, alternatives where the user has to select exactly one of multiple features, are only used in the I/O subsystem. Its alternatives are shown in Figure 3.11. The absence of alternatives, which

[9]The classic tree representation of the feature diagram would not fit onto a single page, therefore a star-like layout has been chosen. Still the diagram shows a tree structure with 'Berkeley DB' as the tree's root.

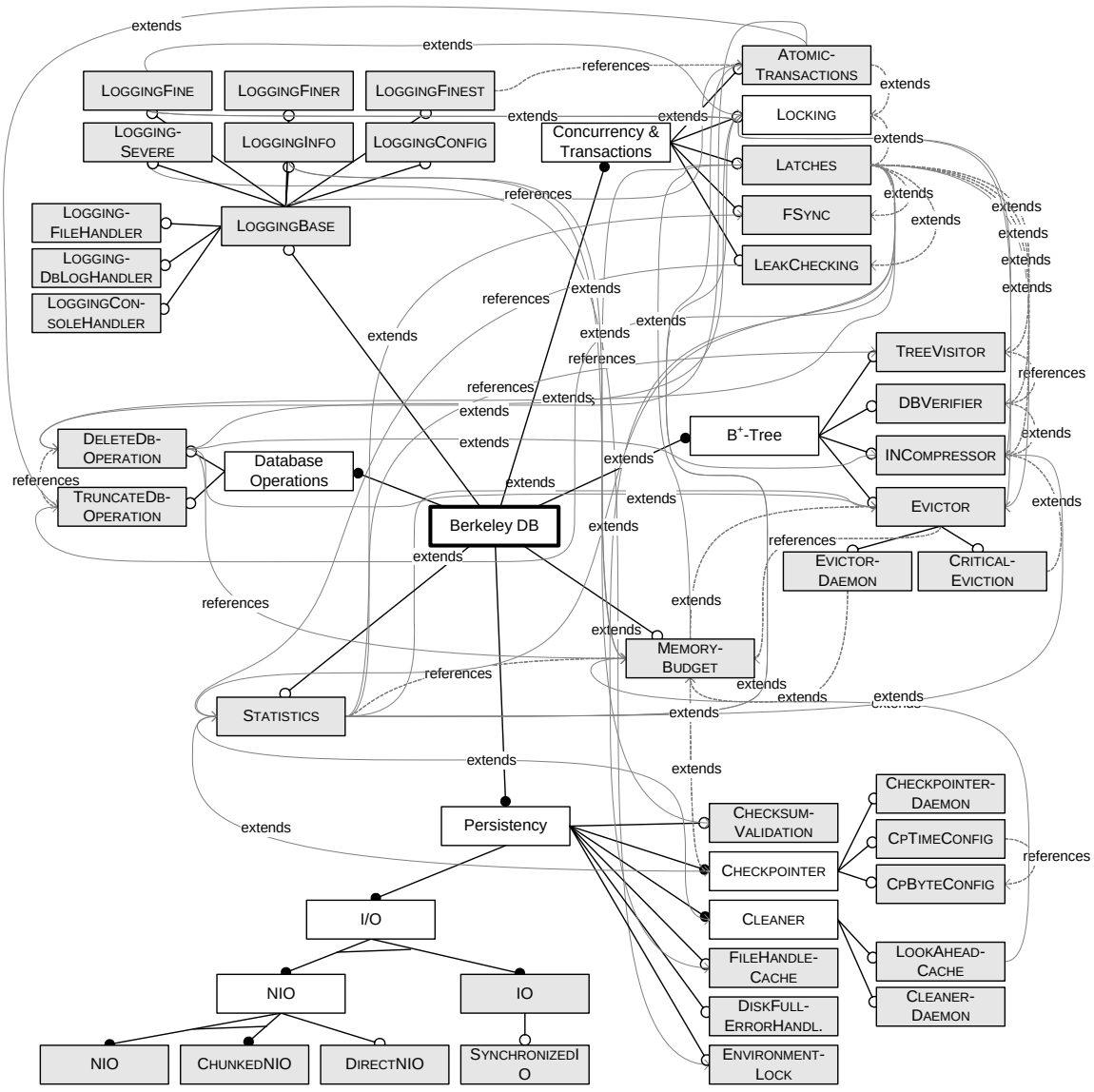


Figure 3.8: Feature Diagram of Berkeley DB.

```
1  BerkeleyDb : [FLogging] Persistency [*@\featureStatistics*@] [MemoryBudget] Concurrency
       DbOperation* Btree BASE;
2
3  Concurrency
4      : Latches
5      | FSync
6      | AtomicTransactions
7      | Locking
8      | LeakChecking;
9
10 Logging : [LoggingFileHandler] [LoggingConsoleHandler] [LoggingDbLogHandler] [Logging-
       Finest] [LoggingFiner] [LoggingFine] [LoggingInfo] [LoggingConfig] [LoggingSevere]
       LoggingBase;
11
12 Persistency : PersistencyFeatures* IOFeature;
13
14 IOFeature
15     : [DirectNIO] NIOType
16     | [SynchronizedIO] IO;
17
18 NIOType
19     : NIO :: NotChunked
20     | ChunkedNIO;
21
22 PersistencyFeatures
23     : ChecksumValidation
24     | FileHandleCache
25     | DiskFullErrorHandling
26     | EnvironmentLock
27     | [CpTimeConfig] [CpBytesConfig] [CheckpointerDaemon]
28     | [LookAheadCache] [CleanerDaemon];
29
30 Btree : [DbVerifier] [TreeVisitor] [INCompressor] [Evictor];
31
32 Evictor : [CriticalEviction] [EvictorDaemon] Evictor;
33
34 DbOperation
35     : DeleteDbOperation
36     | TruncateDbOperation;
37
38 %% //constraints
39 Evictor or EvictorDaemon or LookAheadCache or Statistics implies MemoryBudget;
40 LeakChecking implies Statistics;
41 CriticalEviction implies INCompressor;
42 CpTimeConfig implies CpBytesConfig;
43 DeleteDbOperation implies Locking and Evictor and INCompressor and MemoryBudget;
44 Latches implies Locking and LeakChecking and DeleteDbOperation and Evictor and FileHandle-
       Cache and FSync and INCompressor and MemoryBudget and Statistics and TreeVisitor and
       TruncateDbOperation and DbVerifier;
45 LoggingSevere implies EnvironmentLock;
46 LoggingFine implies Locking and Evictor and INCompressor;
47 LoggingInfo implies ChecksumValidation and MemoryBudget;
48 LoggingBase or LoggingFinest implies AtomicTransactions;
49 MemoryBudget implies Evictor and Latches;
50 Statistics implies Locking and Evictor and FSync and INCompressor and AtomicTransactions
       and TreeVisitor;
51 AtomicTransactions implies Locking and DeleteDbOperation and TruncateDbOperation;
52 TruncateDbOperation implies DeleteDbOperation;
53 DbVerifier implies INCompressor and TreeVisitor;
```

Figure 3.9: *guidsl* Grammar of Berkeley DB.

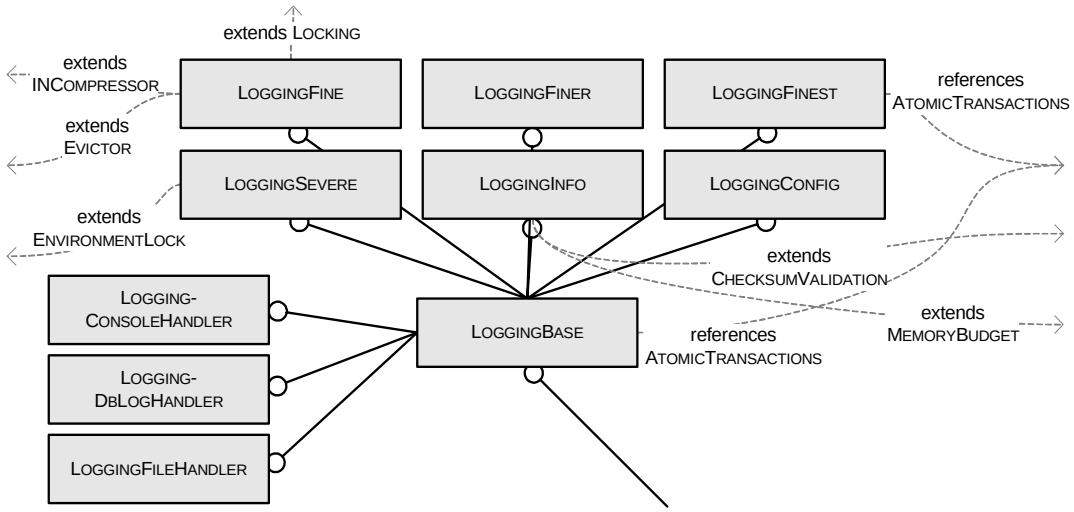


Figure 3.10: Logging Subsystem Feature Diagram.

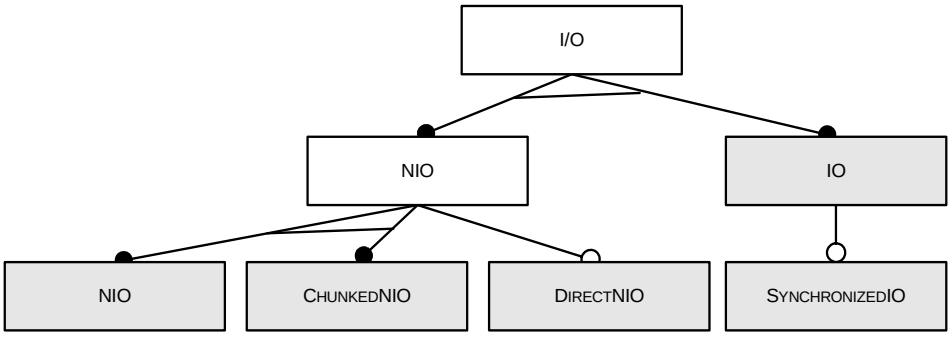


Figure 3.11: I/O Subsystem Feature Diagram.

are otherwise common to SPLs, can be explained with the fact that an existing application was refactored. Only in those rare cases where the original application already had different alternative implementation for the same conceptual feature, alternative features can be expected in a feature model. In Berkeley DB the I/O access methods were implemented in three different ways. The user can chose which operation to use with parameters. In the other cases there was only one implementation for each feature, as expected in a normal application. Nevertheless, the refactoring into features lays the foundations for developers to write alternative features to the refactored ones.

Finally, we can see many dependencies in the feature model. All in all, 62 % of the refactored features require other features, either because they reference them, or because they extend them. Furthermore these dependencies can be categorized as semantic dependencies or as implementation dependencies. Semantic dependencies occur because certain features only make sense in combination with other features. For example CRITICALEVICTION can only be executed when the EVICTOR feature is available. In contrast, implementation dependencies only occur because of the feature's implementation. A simple example is the TRUNCATEDBOPERATION feature that internally uses the delete operation from the DELETEDBOPERATION feature for its implementation; it deletes and recreates the database instead of deleting every entry individually. Another less obvious example is the STATISTICS feature that requires seven other refactored features including EVICTOR, MEMORYBUDGET, TREEVISITOR, and LOCKING, because it collects data about all these other—now optional—features of the system.

Table 3.3 lists all dependencies between in the refactored and marked features, divided by these categories. It shows that there are far more implementation dependencies than semantic dependencies, and while semantic dependencies are usually caused by references, and most of them are already covered by the parent–child relationships in the tree model, implementation dependencies usually exist because one feature extends others. Furthermore it shows that on the implementation level some features i.e., LATCHES, LOGGING, MEMORYBUDGET, DELETEDBOPERATION, and STATISTICS, influence many others or are required by many others.

During the refactoring described in this chapter we made several observations. In the following four chapters we discuss the most important of those. First we analyze the expressiveness of the AspectJ language and discuss the possibilities and limitations of refactoring features with AspectJ from a technical perspective. Afterwards, we discuss the order of refactoring and feature composition. Third, we analyze the feature dependencies and discuss the feature optionality problem, before finally, we focus on readability and maintainability of the resulting code.

	Reference	Extensions
Semantic dependency	CPBYTESCONFIG ⇒ CHECKPOINTER CPTIMECONFIG ⇒ CHECKPOINTER DIRECTNIO ⇒ NIO ∨ CHUNKEDNIO EVICTOR ⇒ MEMORYBUDGET EVICTORDAEMON ⇒ EVICTOR SYNCHRONIZEDIO ⇒ IO	CRITICALEVICTION ⇒ EVICTOR ATOMICTRANSA. ⇒ LOCKING DBVERIFIER ⇒ INCOMPRESSOR
Implementation dependency	CPTIMECONFIG ⇒ CPBYTESCONFIG DELETEDBO. ⇒ MEMORYBUDGET LATCHES ⇒ MEMORYBUDGET LEAKCHECKING ⇒ STATISTICS LOGGINGBASE ⇒ ATOMICTRANSA. LOGGINGFINEST ⇒ ATOMICTRANSA. STATISTICS ⇒ MEMORYBUDGET STATISTICS ⇒ TREEVISITOR TRUNCATEDBO. ⇒ DELETEDBO. DBVERIFIER ⇒ TREEVISITOR	CHECKPOINTER ⇒ MEMORYBUDGET CRITICALEVICTION ⇒ INCOMPRESSOR DELETEDBOPERATION ⇒ EVICTOR DELETEDBOPERATION ⇒ INCOMPRESSOR DELETEDBOPERATION ⇒ LOCKING EVICTORDAEMON ⇒ MEMORYBUDGET LATCHES ⇒ DELETEDBOPERATION LATCHES ⇒ EVICTOR LATCHES ⇒ FILEHANDLECACHE LATCHES ⇒ FSYNC LATCHES ⇒ INCOMPRESSOR LATCHES ⇒ LEAKCHECKING LATCHES ⇒ LOCKING LATCHES ⇒ STATISTICS LATCHES ⇒ TREEVISITOR LATCHES ⇒ TRUNCATEDBOPERATION LATCHES ⇒ DBVERIFIER LOGGINGINFO ⇒ CHECKSUMVALIDATION LOGGINGSEVERE ⇒ ENVIRONMENTLOCK LOGGINGFINE ⇒ EVICTOR LOGGINGFINE ⇒ INCOMPRESSOR LOGGINGFINE ⇒ LOCKING LOGGINGINFO ⇒ MEMORYBUDGET LOOKAHEADCACHE ⇒ MEMORYBUDGET MEMORYBUDGET ⇒ EVICTOR MEMORYBUDGET ⇒ LATCHES STATISTICS ⇒ EVICTOR STATISTICS ⇒ FSYNC STATISTICS ⇒ INCOMPRESSOR STATISTICS ⇒ LOCKING STATISTICS ⇒ ATOMICTRANSA. ATOMICTRANSA. ⇒ DELETEDBO. ATOMICTRANSA. ⇒ TRUNCATEDBO.

Table 3.3: Feature Dependencies.

Chapter 4

Expressiveness of AspectJ Language Constructs

During the refactoring process we reached the limits of AspectJ several times and had to express refactorings in ways we consider unintuitive or inelegant. While most of the code could be refactored into features using the refactorings described in Section 3.3, we found pieces of code that were not so easy to refactor and that required major changes to the original source code. In this section we describe which language constructs and their limits and we discuss possible improvements.

4.1 AspectJ Language Constructs

In order to discuss the expressiveness of AspectJ we examine which language constructs were used. Our evaluation extends the evaluation criteria proposed by Apel et al. (2006a): (1) We focus on the heterogeneous extensions static introductions and method refinements, (2) we continue with the advanced concepts of weaving restrictions, call pointcuts, and conditional extensions that are special to AspectJ, (3) finally, we examine AspectJ's homogeneous extensions.

4.1.1 Static Introductions

Static introductions are simple constructs to introduce new interfaces, classes, methods, or member variables as shown in Section 3.3.1. We used static introductions frequently during our refactorings. Overall we introduced 4 interfaces, 58 classes, 365 methods and 213 member variables. Compared to overall 484 dynamic extensions discussed below, the majority of all extensions are static.

We did not need special statements like *declare error*, *declare soft*, or *declare parents*, which are unique to AspectJ.

4.1.2 Method extensions

A basic dynamic extension is a unconditional method extension. As described with the ESMR refactoring in Section 3.3.2, this extension is done with AspectJ by advising an *execution join point*.

During refactoring Berkeley DB, basic dynamic extensions have been used 214 times, in 44 % of all dynamic extensions. The low percentage already indicates that this basic extension is not sufficient for all kinds of refactorings. We needed frequently extensions with a finer granularity than extending whole methods.

4.1.3 Call extensions

Call pointcuts can be used to extend all calls to a target method. This extension is similar to a method extension of an execution join point described above, but exposes additional information about the caller.

It is possible to restrict the *call* pointcut with other pointcut designators like *within* or *withincode*, to extend not all calls, but only calls located in certain classes or methods. Such restrictions to extend only calls in one method can be used to emulate the extension of individual statements inside a method. For example, when a call inside a method should be synchronized it is possible to extend this call, not the entire method, with a *call* and *withincode* pointcut combination. Such statement extension emulations are discussed later. We used them frequently (54 times) during refactoring.

At the beginning of our refactorings, we often used call and execution interchangeably. However, we rarely used *call* pointcuts where they could not be replaced with execution pointcuts and where they were not used to emulate statement extensions. In these few cases the additional information about the caller is needed. Call pointcuts are mainly just required for statement extensions.

4.1.4 Conditional extensions

Using the pointcuts *cflow*, *if*, *target*, *this*, or *args*, AspectJ allows users to write conditional extensions, i.e., extensions that are executed only when a runtime condition holds. Using *cflow* pointcuts it is possible to extend a join point only when it occurs in the control flow of another join point. Using the *if* pointcut extensions are executed only when a predefined condition hold, e.g., a value is not null. Finally, the pointcuts *target*, *this*, and *args* can be used to execute extensions only when an object satisfies a type condition.

All these conditional pointcuts, except the *cflow* pointcut, can be replaced with conditions inside the advice as shown in Figure 4.1. The *if* pointcut can be replaced with an *if* statement with the same condition inside the advice and the *target* pointcut can be replaced with an *instanceof* condition. We prefer the unconditional version because we find it easier to read. We therefore never use a conditional *if* pointcut and only used a conditional *target* pointcut in few cases for a special problem explained later in Section 4.2.5.

```
1  aspect EvictorDaemon {
2    after(MemoryBudget mb) :
3        execution(void MemoryBudget.updateMiscMemoryUsage(long)) && this(mb)
4        && if(mb.getCacheMemoryUsage() > mb.cacheBudget) {
5      mb.envImpl.alertEvictorDaemon();
6    }
7    ...
8  }
9  aspect TxnBudget {
10   after(Txn txn):
11     call(boolean Set.add(Object)) && target(ReadLocksSet) && this(txn) {
12     txn.updateMemoryUsage(READ_LOCK_OVERHEAD);
13   }
14 }
```

(a) Conditional extensions with *if* and *target* pointcuts.

```
1  aspect EvictorDaemon {
2    after(MemoryBudget mb) :
3        execution(void MemoryBudget.updateMiscMemoryUsage(long)) && this(mb) {
4      if (mb.getCacheMemoryUsage() > mb.cacheBudget) {
5        mb.envImpl.alertEvictorDaemon();
6      }
7    }
8    ...
9  }
10 aspect TxnBudget {
11   after(Txn txn, Set set):
12     call(boolean Set.add(Object)) && target(set) && this(txn) {
13     if (set instanceof ReadLocksSet) {
14       txn.updateMemoryUsage(READ_LOCK_OVERHEAD);
15     }
16   }
17 }
```

(b) Unconditional extension with inner *if* and *instanceof* conditions.

Figure 4.1: Comparison of Conditional and Unconditional Crosscuts.

The pointcut *cflow* was never used to express a conditional extension. We tried to use it for statement extension emulation but found that the *withincode* statement could be used similarly, but without depending on runtime conditions, which is more efficient. Furthermore, the absence of *cflow* pointcuts can be explained by our refactorings, because there is no equivalent expression to *cflow* in the original object-oriented code. Consequently, while refactoring legacy code, we did not find conditions that depend on the control flow, because this was already solved differently using an object-oriented design. However, it might be possible to restructure the refactored code to use *cflow* pointcuts instead of other potentially more spacious extensions, although these changes are complex and a thorough understanding of the source code. We have not found such a case in our refactorings.

In summary, conditional extensions seem not useful for implementing features. This is remarkable since conditional extensions are unique constructs of AspectJ that are not supported by most other languages. Hence we expected some benefits.

4.1.5 Exceptions

The exception handling of an application can be extended by AspectJ. AspectJ provides the *handler* pointcut to capture exceptions and the possibility to advise methods only when they throw an exception, e.g., *'after() throwing (IOException e) : {...}'*. Furthermore, advice statements can catch exceptions or throw new ones.

In our refactorings we only tried to change the exception handling once. In the DISKFULLERRORHANDLING feature we catch an exception and execute a special strategy to save data in case the disk space runs out. On the basis of a single feature we cannot discuss exception handling with AspectJ. However, we observed a problem when extending join points with code that can throw exceptions that we discuss later.

4.1.6 Homogeneous extensions

The ability to create homogeneous extensions is one of the key concepts that distinguishes AOP from other technologies that could be used to implement features. Using AOP it is possible to quantify over multiple join points to extend them with identical code.

First of all, AspectJ allows homogeneous static introductions, however, we did not need them. We presume the reason for the absence is that object-oriented concepts like inheritance already avoided repeated member variables or methods in the object-oriented design of Berkeley DB.

Second, AspectJ allows various homogeneous dynamic extensions. We tried to apply these homogeneous extensions during refactoring when we observed scattered or replicated code. For example, the extension taken from the LATCHES feature shown in Figure 4.2 uses two homogeneous pointcuts (Lines 8–17) to extend multiple methods with the identical synchronization code.

Because there are different possibilities to define homogeneous extensions which each have their individual problems we categorize homogeneous extensions into three groups:

- **Pattern Expressions.** The first group uses lexical pattern expressions to quantify multiple join points. With these pattern expressions it is possible to extend multiple methods or even methods from different classes. For example, the advice statement with the pointcut *'execution(void MemoryBudget.update*MemoryUsage(..))'* from the EVICTORDAEMON feature extends the three methods *updateTreeMemoryUsage*, *updateMiscMemoryUsage*, and *updateLockMemoryUsage* to wake the Evictor daemon after each memory update. However, in Berkeley DB only 7 advice statements use pattern expressions for homogeneous extensions and mostly only two methods are extended.

- **Explicit Enumeration.** The second group are extensions where multiple join points are advised but they are specified explicitly and connected with the '||' operator, like the pointcuts *latchedWriteMethods* and *latchedReadMethods* in Figure 4.2 (Lines 8–17). These explicit homogeneous extensions were used 25 times in our refactorings, however rarely to quantify more than two join points.

```
1  public aspect TriggerLatchExtension {
2    private SharedLatch EnvironmentImpl.triggerLatch;
3
4    after(EnvironmentImpl envImpl) :
         execution(EnvironmentImpl.new(File, EnvironmentConfig)) && target(envImpl) {
5      envImpl.triggerLatch = LatchSupport.makeSharedLatch("TriggerLatch", envImpl);
6    }
7
8    pointcut latchedWriteMethods(Database db) : (
9        execution(void Database.addTrigger(DatabaseTrigger, boolean)) ||
10       execution(void Database.removeTrigger(DatabaseTrigger)) ||
11       execution(void Database.removeAllTriggers())
12     ) && target(db);
13
14   pointcut latchedReadMethods(Database db) : (
15       execution(List Database.getSecondaryDatabases()) ||
16       execution(void Database.notifyTriggers(Locker, DatabaseEntry, DatabaseEntry,
             DatabaseEntry))
17     ) && target(db);
18
19   Object around(Database db) throws DatabaseException :  latchedWriteMethods(db) {
20     SharedLatch triggerLatch = db.envHandle.getEnvironmentImpl().triggerLatch;
21     triggerLatch.acquireExclusive();
22     try {
23       return proceed(db);
24     } finally {
25       if (db.triggerList.size() == 0) {
26         db.triggerList = null;
27       }
28       triggerLatch.release();
29     }
30   }
31
32   Object around(Database db) throws DatabaseException : latchedReadMethods(db) {
33     if (db.hasTriggers()) {
34       SharedLatch triggerLatch = db.envHandle.getEnvironmentImpl().tiggerLatch;
35       triggerLatch.acquireShared();
36       if (db.triggerList == null) {
37         db.triggerList = new ArrayList();
38       }
39       try {
40         return proceed(db);
41       } finally {
42         triggerLatch.release();
43       }
44     } else
45       return proceed(db);
46   }
47 }
```

Figure 4.2: Homogeneous Crosscuts for the Trigger Latch Extension.

- **Homogeneous Statement Extensions.** Finally, the third group are *call* pointcuts that implement homogeneous statement extensions. There are two typical cases when this extension is used: (1) A method is extended only when it is called from certain methods, thus a typical *call* and *within* pointcut combination is used to emulate statement extensions. (2) An method extension needs information about the caller. For example, every call to a transactions lock method must be synchronized in the cursor that calls this lock method. Therefore the advice needs a reference of the cursor object.

	# join points				
Category	2	3	4	>4	$\sum$
Pattern expressions	5	1	1	0	7
Explicit enumeration	15	7	1	2	25
Homog. statement extensions	11	4	2	3	20

Table 4.1: Homogeneous Dynamic Extensions.

In Table 4.1 we show the number of homogeneous expressions by category and by number of extended join points in Berkeley DB. The total number of homogeneous extensions (52) is surprisingly low compared to 1072 heterogeneous extensions. In some cases technical problems like the *Exception Introduction Problem* or the *Statement Extension Problem*, prevented the use of homogeneous pointcuts, even for prototypical examples that are similar to those described in AspectJ literature (e.g., Laddad, 2003). For example, the LATCHES feature uses very similar code—inside one class often even repeated code—to synchronize different code fragments. However, we advised whole methods that allow easy homogeneous extensions only in few cases and frequently faced problems that required workarounds. Accordingly different pointcuts were needed, including EIMR refactorings or hook methods to extend the base code, what often prevented a homogeneous extension. One example where a language problem prevented a homogeneous extension is the latching code shown later in Figure 4.7 (p. 70), where the *Exception Introduction Problem* prevents a call pointcut and the subsequent unsynchronized code prevents an execution pointcut (cf. Section 4.2.3, p. 68). Therefore, the lexical pattern expressions—described often in AspectJ literature to capture join points from multiple methods—are rarely used, only in 7 advice statements. Additionally, code scattering has already been avoided by the object-oriented design, so that the few cases where pattern expressions could be used never advised more than 4 join points.

Furthermore, the benefit of homogeneous extensions is questionable. The scattered and repeated code fragments are usually very small, in most cases it is one single call. Thus often the code required to reintroduce these calls or match the join points, especially with explicit enumerated pointcuts, is larger than the code size saved by removing the repetition.

Finally, most homogeneous extensions usually only extend join points inside one class. Only 7 homogeneous enumerated or statement extensions extended join points in two different target classes. This again indicates the low complexity of our homogeneous extensions.

We have no single extension that crosscuts the whole application.

An interesting side effect we observed is that during the early refactorings we used pattern expressions in almost all pointcuts, even though we did not use them for homogeneous extensions, but to simplify the writing of pointcuts. For example, instead of the long full signature *'execution(void Database.notifyTriggers(Locker, DatabaseEntry, DatabaseEntry, DatabaseEntry))'* we just used *'execution(* Database.notifyTriggers(..))'*. We often omitted the return type and parameters when we did not face overloaded version of the method with different return values or parameters that would require the exact specification. However, as we discuss later, these patterns are potentially dangerous during maintenance and evolution.

Overall, we suspect that replicated, scattered code is mainly a problem of poorly designed or grown legacy applications and can already be avoided to a large extent. However, once the statement extension and exception introduction problems are solved homogeneous expressions might still be a good help in some cases, even though we think that it only will be used for a small fraction of all extensions as also indicated by recent research (Apel et al., 2006a; Apel and Batory, 2006).

4.1.7 Other language constructs

Other than the constructs previously discussed, AspectJ provides possibilities to structure aspects themselfs. Aspects can be abstract and part of an inheritance hierarchy. Also explicit pointcut definitions can be created to reuse pointcuts.

During our refactorings, we could not reuse any aspect implementations and therefore did not use aspect inheritance. Similarly most advice declarations use anonymous pointcuts that are not reused. We started with using only explicit pointcut but soon changed to anonymous ones because it reduces code size and is more convenient do develop. Only in 20 of 484 cases explicit pointcuts were used, partly because the anonymous pointcuts became too complex and partly because pointcuts were used by two advice declarations. Additionally, all explicit pointcuts were used locally, i.e., no advice referred to a pointcut from a different aspect. The reused pointcuts usually occur in combination with more advanced statements to emulate statement extensions.

This shows again that most of the advanced language constructs are not needed for refactoring, and the basic extension mechanisms are used most.

4.2 Language Limitations

In the previous sections we discussed how existing constructs were used during our refactorings. Unfortunately we could not express not all problems with the existing AspectJ constructs. Sometimes we had to change the source code or use workarounds because of certain limitations of the AspectJ language. In this section we present the problems observed during refactoring and discuss possible solutions.

4.2.1 Statement Extension Problem

The problem that occurred most frequently and increased the required effort for refactoring is AspectJ's inability to extend individual statements inside a method. As already shown for the ECR and EIMR refactorings in Section 3.3.2, it is often necessary to extend a program at (before, after, around) an individual statement or a sequence of statements inside a method. AspectJ's join point model does not provide join points at statement level, but some advanced pointcuts can be used to emulate statement extensions:

Mostly, we used *call join points* enable allow to quantify calls to a method, that are situated inside other methods. Although AspectJ allows only to quantify all calls to a target method at once, it is possible to narrow it down to all call join points inside a single method using the *withincode* pointcut as shown in Figure 3.4 (p. 36). Hence, call join points can be used to extend statements inside a method, under the condition that the statement is a call statement and that it is the only call to the target method inside this method.

Similar to call join points, *get and set join points* are exposed at every access to member variables and can be used to emulate statement extensions as well. Apart from call, set and get join points, AspectJ does not provide any language constructs to extend individual statements. So, only single method calls or access to member variables that are unique inside a method can be extended. To extend a sequence of statements still workarounds like the Extract Inner Method Refinement refactoring, or hook methods are necessary.

We used the *call* and *withincode* pointcut combination 54 times to emulate statement extensions and used 172 extensions to hook method where statement extensions could not be emulated. Compared to this, 214 extensions were simple method refinements, where whole methods and not individual statements are extended. These numbers emphasize the importance of statement extensions and explain our impression that the restriction to only emulate statement extensions with AspectJ was a hindering and time consuming factor in our refactorings.

Unfortunately, we know of no other approach that could be used to implement features that supports statement extensions. On the contrary most alternative languages like *Jak* in AHEAD support only simple method refinements, and do not offer any emulation possibilities. From this perspective the emulation of statement extensions in AspectJ is an improvement over many other technologies. However, the emulation is not sufficient. On the one hand, we recognize the problems of statement level pointcuts: They are complex to define and tend to be very fragile and hard to maintain. On the other hand, we obviously need a finer grained possibility to extend code.

The problem stems from the concept of obliviousness in AOP (cf. Section 2.4). It assumes that normal programs provide enough join points, so that explicit join points in a prepared code, like explicit hook methods, are not required (*"Just program like you always do, and we'll be able to add the aspects later"* (Filman and Friedman, 2005)). Our observations show that this is clearly not the case for Berkeley DB: the existing join points are not sufficient. We frequently need to provide explicit join points inside a method by creating hook methods. Thus, it is necessary to design for extensibility.

One possible solution to create explicit join points that avoids hook methods was pro-

posed by Eaddy and Aho (2006). They suggest to use *statement annotations* to create statement level join points. This way, the developer can annotate arbitrary statements for later extension from a feature. Statement annotations are similar to hook methods—both create explicit variation points for later extension—but have some advantages: First they do not add empty method declarations to the source code, what is claimed to affect design and code quality adversely and to be confusing to the programmer (Murphy et al., 2001). Second, statement annotations add a join point for one specific statement, not a join point before or after this statement like hook methods do (Eaddy and Aho, 2006). Therefore, annotations are less fragile to subsequent source code changes because they expose join points exactly for the advised statements. Compared to the current approach of emulating statement extensions with AspectJ, annotations are more stable and easier to use. An open problem is that statement annotations as proposed by Eaddy and Aho (2006) do not enable to advise sequences of statements, like the one shown earlier in Figure 3.5 (p. 37). We therefore propose to additionally add the possibility to annotate statement sequences or code blocks. Unfortunately, Java currently does not support statement annotations and there is no AspectJ compiler that supports them or similar concepts.

A different approach to solve the Statement Extension Problem is to search for a new way applications are proposed, instead of quantifying over join points. A different approach, our *ColoredIDE*, is discussed briefly in the further work Section 9.2.

In summary, we encountered frequently the Statement Extension Problem during our refactorings. Because no other solution exists currently, we emulated statement extensions with complex AspectJ constructs or hook methods. However, we argue that better solutions are needed and support the statement annotations approach to solve the statement extension problem.

4.2.2 Parameter Introduction and Passing Problem

Another problem that occurred during the refactoring of the ATOMICTRANSACTIONS and MEMORYBUDGET features is the change of method signatures to introduce parameters. This is also the problem that most hindered us in refactoring the LOCKING feature and it is expected to occur frequently in further refactoring.

The problem occurs when a parameter of a method belongs to one feature as shown in Figure 4.3. The atomic transaction code is actually rather small: it only decides which locker, other than the non-transactional default locker *BasicLocker* or *ThreadLocker*, to use. However, for this extension, the user can specify a transaction parameter, that is declared in the interface method *'Database.delete'* and passed along to the *getWritableLocker* method. In the shown case the parameter is passed directly through to the second method, in other cases we found the parameter is passed through multiple methods along the control flow until it is used in an internal method.

We could easily refactor the transaction code in *getWritableLocker* with an Extract Simple Method Refinement refactoring, but this would leave the transaction parameters inside the code. Thus, even when the ATOMICTRANSACTIONS feature is not included, the parameter is present for both, interface methods and internal methods, and it is passed along

```
1  public class Database {
2    public OperationStatus delete(Transaction txn, DatabaseEntry key)
         throws DatabaseException {
3      trace(Level.FINEST, "Database.delete", txn, key, null, null);
4
5      Locker locker = LockerFactory.getWritableLocker(envHandle, txn, isTransactional());
6      return deleteInternal(locker, key);
7    }
8  }
9  public class LockerFactory {
10   public static Locker getWritableLocker(Environment env, Transaction userTxn, boolean
         dbIsTransactional) throws DatabaseException {
11     if (dbIsTransactional && userTxn == null) {
12       return new AutoTxn(env);
13     }
14     if (userTxn != null) {
15       return userTxn.getLocker();
16     }
17     if (retainNonTxnLocks) {
18       return new BasicLocker(env);
19     } else {
20       return new ThreadLocker(env);
21     }
22   }
23 }
```

Figure 4.3: Example of the Parameter Introduction and Passing Problem.

inside the application. For a full refactoring of the method we also want to remove and reintroduce the parameters, because it semantically belongs to the feature and not to the base code[1].

AspectJ does not support any changes of method signatures and consequently also does not allow to introduce new parameters. Thus, refactoring parameters that belong to a feature are a major problem.

We have experimented with different solutions or workarounds for this parameter passing problem. First, we discuss to leave the parameters in the code anyway, second, we show an approach to pass parameters with a *cflow* pointcut, third, we tried to use general parameter objects. Finally, we discuss a potential extension to AspectJ.

Leave Parameters in Base Code. The easiest solution is to leave the parameters in the base code. When objects are passed like the *Transaction* object, an empty class can be provided by the base code, that is later extended by the ATOMICTRANSACTIONS when needed. The base code simply passes instances of the empty *Transaction* class or *null* values.

However we do not consider this as a good approach, because some logic of the feature

[1]Note: A similar problem occurs with return values of methods. For some methods most code belongs to the base code, but the return value belongs to a feature. For example, the methods to evict memory of the EVICTOR feature return the amount of evicted memory. Yet, when the MEMORYBUDGET feature is not included this information is not necessary and cannot even be calculated, the method therefore currently just returns an empty value. For a clean decomposition, it should be possible to change the return value as well as the parameters.

```
1  public aspect AtomicTransaction {
2    public OperationStatus Database.delete(Transaction txn, DatabaseEntry key)
         throws DatabaseException {
3      delete(key);
4    }
5    pointcut deleteInterface(Transaction txn) :
         execution(OperationStatus Database.delete(Transaction, DatabaseEntry)) && args(txn,
         *);
6    Locker around (Transaction userTxn) :
         execution(Locker LockerFactory.getWritableLocker(Environment)) &&
         cflow(deleteInterface(userTxn)) {
7      if (userTxn != null) {
8        return userTxn.getLocker();
9      }
10     proceed(userTxn);
11   }
12 }
```

Figure 4.4: Passing a Parameter with a *cflow* Pointcut.

must remain in the base code, e.g., the logic which parameter is needed where and must be passed where. This way, many parameters are left in the base code which are never needed unless a certain feature is included. This approach produces especially bad code when alternative features are used with use different parameters, e.g., multiple different implementations of the ATOMICTRANSACTIONS feature. Now every parameter that is ever potentially used by any of those features must be included in the source code. This is not satisfactory for our decomposition into features.

We strongly argue against leaving parameters in base code when refactoring an application into features that should be used productively, even though it seems to be commonly used because no other simple solution is available.

***cflow* Pointcut.** With AspectJ we can pass a value from one method to an advice without using a parameter, but by using the *cflow* pointcut. For this we declare a pointcut that captures a parameter from a method and we declare a second pointcut in the control flow of the first one. We can now advise the join points of the second pointcut while we can still access the parameters from the first pointcut from our advice statement. It is even possible to use this approach with a homogeneous pointcut that captures a parameter from multiple methods.

Figure 4.4 shows such an implementation for our earlier delete operation example: First, a new interface method with the transaction parameter is introduced statically (Lines 2–4), then this interface method is captured by the *deleteInterface* pointcut (Line 5). Finally, the *getWritableLocker* method is advised, with the second anonymous pointcut that uses the control flow statement to access the transaction parameter from the interface method (Lines 6–10). This way, the advice body can use the parameter that was previously declared only in the new interface method. Node: The *deleteInterface* pointcut could also be specified homogeneously for all interface methods that support a transaction parameter.

This solution seemed promising, but when we implemented the ATOMICTRANSACTIONS feature using the cflow approach, we discovered several problems:

1. We regard this approach as unintuitive and hard to maintain for developers familiar with procedural or object-oriented programming. It hides how the parameter is passed in the application and passes the parameter 'magically' along.

2. We cannot prevent that the old interface method is called, except by using a second advice that throws an exception every time the old interface method is called outside the control flow of the new interface method. This requires another cflow pointcut and increase the complexity. Alternatively, a *declare error* statement could be used to forbid access to the old method, although this statement becomes similarly complex because it needs a *withincode* pointcut to exclude the call from the new interface method.

3. It is not possible to call the inner method with a different parameter than passed by the user. For example, the delete method could not request a non-transactional locker to perform some service operations. This can be especially a problem when service threads like the Cleaner or Evictor are invoked inside the control flow. In deeply nested control flows it might even be hard to detect those cases, because the parameter is no longer visible in method signatures.

4. The aspect becomes very complex, especially when different parameters are involved. In Figure 4.4 only the transaction parameter was reintroduced to the *getWritable-Locker* method. Extra effort would be required to capture the *isTransactional* parameter and to reintroduce the transaction parameter into the trace method to complete the example from Figure 4.3. The LOCKING feature passes to many parameters that we did not attempt a refactoring because the resulting code would be to complex, unreadable, and unmaintainable (also cf. Section 7).

5. Finally, feature interactions (cf. Chapter 5) become very complicated. When more than one feature introduce parameters to a method every possible combination of interface methods must be provided. Consider a potential feature that introduces user authentication with a user permission object to the delete operations. Now there are four different versions of the delete methods: *'delete(key)'*, *'delete(transaction, key)'*, *'delete(permission, key)'* and *'delete(transaction, permission, key)'*. For each interface method the *cflow* pointcut must be implemented differently. The number of necessary interface methods and *cflow* implementations grows exponentially with the number of features that extend a method[2].

Therefore, we do not consider cflow constructs as a good approach to pass parameters. It is technically possible but the problems, especially the growing complexity, forbid their use.

[2]To be precise, using pattern expressions it is possible to address the first and the last parameter, so that the cflow implementation can be reused. For example, the following pointcut captures the transaction parameter in all delete methods that have it as first parameter: e.g., *'pointcut deleteInterface(Transaction txn) : execution(OperationStatus Database.delete(Transaction, ..)) && args(txn, ..);'*. However, approach does not scale for more than two features because the AspectJ pattern mechanism forces that the parameter always occurs at a fixed position from the beginning or the end, e.g., first, second, third, last, second last, etc.

```
1  public class OperationContext {}
2  public class Database {
3    public OperationStatus delete(OperationContext context, DatabaseEntry key)
         throws DatabaseException {
4      trace(Level.FINEST, "Database.delete", context, null, null);
5      Locker locker = LockerFactory.getWritableLocker(context, envHandle);
6      return deleteInternal(locker, key);
7    }
8  }
9  public aspect AtomicTransaction {
10   public Transaction OperationContext.txn;
11   Locker around (OperationContext context) :
         execution(Locker LockerFactory.getWritableLocker(OperationContext, Environment)) &&
         args(context, *) {
12     if (context.txn != null) {
13       return context.txn.getLocker();
14     }
15     proceed(context);
16   }
17   public OperationStatus Database.delete(Transaction txn, DatabaseEntry key)
         throws DatabaseException {
18     OperationContext context = new OperationContext();
19     context.txn=txn;
20     delete(context, key);
21   }
22 }
```

Figure 4.5: Passing Parameters with a Parameter Object.

Parameter Objects. A different approach to handle the parameter passing problem is to use extensible parameter objects. This approach was inspired by the observations made by Lopez-Herrejon and Batory (2001) that show that most parameters are only used as a form of optimization while an extensible design would pass objects. Additionally, this approach was commonly used when extending parameters with AHEAD.

In our example, it is possible to argue that transactions, user permissions, or other values a user must specify additionally to the key are just context information for the operation and can therefore be handled by an *OperationContext* object. When we applied this idea to the above delete operation example we got the source code shown in Figure 4.5. The base code (Lines 1–9) contains an empty context parameter, that is extended and used by the *AtomicTransaction* aspect (Lines 10–23). This aspect introduces the *txn* member variable into the *OperationContext* class (Line 11) and uses this member variable in an extension of the *getWritableLocker* method (Lines 12–17). Note that the new delete method that reintroduces the old interface (Lines 18–22) is only there for the user's convenience, it is possible to use the base function directly when the operation context object has reasonable default values.

This approach is already used in parts of Berkeley DB where many parameters of a method can be configured with an object. For example, the *Environment.openDatabase* method has a *DatabaseConfig* parameter that encapsulates 11 parameters. However, for the ATOMICTRANSACTIONS or LOCKING features we would need parameter objects much more often.

Compared to the cflow approach, parameter objects have several advantages. First,

they do not hide parameters and they are easy to understand for developers familiar with object-oriented programming. The second and more important advantage is that complexity does not grow when more parameters are added. For example, a potential user permission feature could extend the same context object. Each feature would implement only its own extensions, only the number of interface methods, we reintroduced for convenience reasons, would still grow exponentially. Anyway they just encapsulate the creation of the context objects while the actual feature implementation, e.g., the advice in Figure 4.5, remains unchanged.

We implemented a second version of the ATOMICTRANSACTIONS feature with parameter objects as well and found several problems:

1. This approach requires larger object-oriented refactorings of the application to use parameter objects instead of individual parameters. These required enabling refactorings induce a high effort before the refactoring into features can be even started.

2. Not every method can use the same context parameter. In Berkeley DB all interface methods have the transaction parameter and could support a user permission parameter, however some interface methods have additional parameters. For example, the *get* method, that retrieves a value from the database, has an additional *lockMode* parameter to specify the level of isolation (cf. Section 2.1). The *lockMode* also belongs to the ATOMICTRANSACTIONS feature and could be included in the *get* method's context object, however other interface operations, that are not used for retrieval, do not have this value. This shows that there are different context objects needed for different methods. In theory there could be a different context object for each method.

3. A possibility to avoid different context parameters is to have one OperationContext class that contains all parameters that are used by any method. Parameters that are not required for a certain method are simply not read. However, this approach hides the information which parameters are really needed by a method. Workarounds like describing used parameters in a comment or checking the required parameter at the beginning of every method cannot counter the disadvantage.

4. When a parameter is passed through different methods not all parameters might be used. Consider the context object from the *get* method with the *lockMode* parameter. Also the *get* method calls the *getWritableLocker* method to create a locker. However, for the creation of a locker the additional *lockMode* parameter is not needed. Thus, even along the control flow unnecessary parameters are passed and the information which parameter is required for each method is hidden. Additionally, some information are calculated only and passed internally and are not provided by the user, e.g., the *locker* object in the LOCKING feature. A potential solution is to also create an own context object for every internal method but this would result in creating context objects and copying required values, e.g., by using the Adapter pattern (cf. Gamma et al., 1995, p. 139ff), all the time. This would create a huge overhead and would decrease source code readability and performance.

5. It is possible to model the different context objects with inheritance or interfaces, reducing the need to copy parameters from one context objects to another, but the resulting context object hierarchy would be hard to maintain and difficult to extend when new features add further parameters to selected methods.

Overall, we found that this generally approach is usable but becomes very complex because of many different objects or hidden information about required parameters. For a complex productive system we estimate that it is better than the cflow approach, but it is still not a desirable solution. We did not refactor the LOCKING feature with this approach neither, because of the high expected effort for preparatory object-oriented refactoring and the creation of suitable context parameters.

AspectJ Extension. After showing that the existing possibilities are not sufficient to solve the parameter passing problem we discuss how AspectJ could be extended. Basically, two things are needed, it must be possible to introduce a new parameter to a method, i.e., change the methods signature, and it must be possible to extend calls in this method to pass the parameter along, and to access the value from advice.

The possibility to change the method signature introduces new problems, because many pointcut matching mechanisms rely on method signatures. For now we ignore these problems to show the basic idea. On the other hand, the required extensions to method calls are possible, but to access the caller's parameters *cflow* pointcuts are necessary with many of the problems described above. We therefore also assume the presence of a possibility to access the caller's parameters with a new *callerargs* pointcut similar to the *args* pointcut for the target method.

Figure 4.6 shows a potential solution for our delete operation example. Two *add parameter* statements are added to introduce the parameters and one *around* advice (Lines 4–6) is used to pass the parameter along, while the second *around* advice (Lines 7–12) implements the actual extension. Note that the '..' pattern in Lines 4, 5, and 7 indicate that other parameters introduced by other features are possible and do not hinder the extension.

The suggested syntax is far from being user-friendly. Its purpose is to indicate the kind of extension that is needed to solve the parameter passing problem while it is kept similar to existing AspectJ constructs for easier explanation. Further research is needed to investigate side effects, to solve problems with the weaving process, and to simplify the syntax. Probably a completely new construct is needed to pass parameters and the way join points are matched must abstract more from the parameters of a method.

In summary, extending AspectJ might solve the parameter passing problem, but it is unclear if changes of this granularity are ever readable or maintainable. We believe the complexity will be very high. Hence, we suggest research in other directions as well, that do not depend on quantifying join points and weaving code. As one approach our *ColoredIDE* suggestion presented in Section 9.2 solves this problem in a completely different way.

```
1  public aspect AtomicTransactions {
2    add parameter(Transaction txn) :
         execution(OperationStatus Database.remove(DatabaseEntry)) ||
         execution(Locker LockerFactory.getWritableLocker(Environment));
3    add parameter(boolean isTransactional) :
         execution(Locker LockerFactory.getWritableLocker(Environment));
4    Locker around (Transaction txn, Environment env) :
         call(Locker LockerFactory.getWritableLocker(Environment)) &&
         withincode(OperationStatus Database.remove(DatabaseEntry)) && callerargs(txn, ..) &&
          args(env) {
5      return LockerFactory.getWritableLocker(env, txn, database.isTransactional(), ..);
6    }
7    Locker around (Transaction userTxn, boolean isTransactional) :
         execution(Locker LockerFactory.getWritableLocker(Environment, ..)) &&
         args(.., userTxn, isTransactional, ..) {
8      if (userTxn != null) {
9          return userTxn.getLocker();
10     }
11     proceed(userTxn, isTransactional);
12   }
13 }
```

Figure 4.6: Passing Parameters with a potential AspectJ Extension.

4.2.3 Exception Introduction Problem

We found another severe problem when we tried to reintroduce code that can throw an exception. It happened when we attempted to move synchronization code from the base code to the LATCHES feature. As explained before, AspectJ does not allow to change method signatures and thus also does not allow to add new throwable exceptions to a method. This means that advice can throw only exceptions that are declared already in the base code or runtime exceptions that do not need to be declared. Similar to the *Parameter Introduction Problem* this implies that all exceptions ever thrown by any feature must already be declared in the source code. As a workaround AspectJ introduces the possibility to 'soften' exceptions, i.e., to encapsulate them in a runtime exception. However, we avoided softened exceptions because we consider them counterintuitive and hard to handle.

The problem is intensified because of the observed semantics of the call pointcut, that behaves exactly like an execution pointcut on the target method when it comes to exceptions. To be able to throw an exception in advice on a call join point the call's target method must already be declared with this exception, even though the exception is declared in the method that contains the call. Similarly, it is not possible to throw any exception in advice on get and set join points, even though the exception is declared in the containing method. This prevents statement extension emulation in many cases when exceptions are involved, especially for calls to JRE classes.

Figure 4.7 illustrates this problem on an simplified example taken from the LATCHES feature. The *acquire* method from the *Latch* class can throw a *LatchException*, a subclass of the declared *DatabaseException*, if the latch is already held. Because of some subsequent code that is not synchronized, it is not possible to use an simple method extension, but it is necessary to advise the statement that clears the list. This statement can be advised by the *call* and *within* pointcut as shown in Figure 4.7b. Unfortunately, it is not possible to

throw an exception in this case, because the signature of the method *clear* of the *SortedSet* class does not declare this exception and—as an API class of the JRE—cannot be changed. Therefore, the only consequence is to use a hook method as shown in Figure 4.7c that can be advised with either an execution or a call pointcut.

To achieve a more intuitive behavior and to enable statement extension emulation and extensions to API classes, we recommend that the exception should be checked against the container method on *call*, *set* and *get* join points. Furthermore, it should be possible to change method signatures to be able to declare exceptions by features, similar to the proposed *add parameter* construct for the parameter passing problem.

4.2.4 Local Variables Access Problem

Accessing local variables is another major problem in AspectJ. Already in Section 3.3.4, we have shown that workarounds are often required to access local variables from advice. To access these variables either object-oriented refactorings are necessary to expose them as member variables or parameters, or hook methods have to be introduced. In this Section we discuss some further advanced approaches and suggest a possible solutions.

All approaches shown in Section 3.3.4 and used during our refactorings have their weaknesses. Member variables hide the fact that the variable is only used locally and make the source code harder to read. The proposed Replace Method with Method Object refactoring (cf. Section 3.3.4) increases source code complexity, even for the base code, just to achieve accessibility from AspectJ advice. Finally, we consider hook methods as bad design as discussed before, because they introduce empty methods in the base code.

There are constructs with which it is possible to counter some of these problems. For example, the first problem that a member variable hides the fact that it is only used locally, which could lead to possible implementation errors, can be technically countered with AspectJ. The AspectJ *declare error* construct enables access to member variables from certain targets only, as shown in Figure 4.8. Nevertheless, we still do not consider this as a reasonable solution for practical refactoring, because it makes the source code even more complex and requires explicit exceptions in the base code for every aspect that should be allowed to access it.

Another special case is when local variables are the parameters of the callers method, as the *locker* variable earlier in the example in Figure 3.7a (p. 40). In this case, and under the condition that the value has not been changed before the extension, it is possible to capture this variable with a *cflow* pointcut as shown in Figure 4.9. However, even though this could have been useful in some cases, we have not included this approach in our list of refactorings, because we consider it too complex and too fragile for productive use. It is not possible to change the variable before the extension, but this fact is hidden completely from the developer which makes it error-prone for subsequent changes. Also the cflow problems shown in Section 4.1.4 apply here as well.

All workarounds we found to access local variables are unsatisfactory. But access to local variables is an important requirement to support statement extensions. Otherwise most statement extensions are not possible, even with the proposed solution of statement

```
1  public class INList {
2    private SortedSet ins =  new TreeSet();
3    public void clear() throws DatabaseException {
4      latch.acquire();
5      ins.clear();
6      latch.release();
7      ...
8    }
9    ...
10 }
```

(a) Original code.

```
1  public class INList {
2    public void clear() throws DatabaseException {
3      ins.clear();
4      ...
5    }
6    ...
7  }
8  public aspect INListExtension {
9    void around(INList inList) throws DatabaseException : call(void SortedSet.clear()) &&
         withincode(void INList.clear()) && this(inList) {
10     inList.latch.acquire();
11     proceed(inList);
12     inList.latch.release();
13   }
14 }
```

(b) Intended Extension cannot throw exception.

```
1  public class INList {
2    public void clear() throws DatabaseException {
3      hook(ins);
4      ...
5    }
6    private void hook(SortedSet ins) throws DatabaseException {
7      ins.clear();
8    }
9    ...
10 }
11 public aspect INListExtension {
12   void around(INList inList) throws DatabaseException :
         execution(void INList.hook(SortedSet)) && this(inList) {
13     inList.latch.acquire();
14     proceed(inList);
15     inList.latch.release();
16   }
17 }
```

(c) Required implementation with hook method.

Figure 4.7: Exception Introduction Problem at *call* Pointcuts.

```
1  declare error :
2    set(int TargetClass.localVariable) || get(int TargetClass.localVariable)
3    && !withincode(void TargetClass.targetMethod(ParameterTypes))
4    && !within(targetAspect) :
5    ''localVariable is a local variable of method targetMethod.'';
```

Figure 4.8: Preventing Access to Member Variable.

```
1  public class LN {
2    ...
3    public long delete(DatabaseImpl database,...,Locker locker) throws DatabaseException {
4      ...
5      entryType = hook_getEntryType(entryType);
6      ...
7    }
8    private LogEntryType hook_getEntryType(LogEntryType entryType) {
9      return entryType;
10   }
11 }
12 public aspect TransactionalLN {
13   pointcut deleteCall(Locker locker) :
14       execution(long LN.delete(..)) && args(.., locker);
15   LogEntryType around(Locker locker) :
16       call(LogEntryType LN.hook_getEntryType(Locker, LogEntryType))
17       && cflow(deleteCall(locker)) {
18     if (locker.isTransactional())
19       return LogEntryType.LOG_DEL_DUPLN_TRANSACTIONAL;
20     else
21       return proceed(locker);
22   }
23 }
```

Figure 4.9: Capture Local Variable from Caller's Parameter.

```
1 public aspect TransactionalLN {
2   before() : call(DeletedDupLNLogEntry.new(LogEntryType, ...)) {
3     if (context.locker.isTransactional())
4       context.entryType = LogEntryType.LOG_DEL_DUPLN_TRANSACTIONAL;
5   }
6 }
```

Figure 4.10: Access to Local Variables with a *context* Keyword.

annotations. It is hard to determine exact numbers of how often local variable access is needed, but we can estimate that most of the 49 hook methods that contain parameters or return values are used to provide access to local variables.

A simple solution that would prevent this problem is to expose local variables at statement join points. Various context information are already available with the *thisJoinPoint* construct, access to the local variables could be granted in a similar way. This allows AspectJ to make very fine grained changes but consequently follows the approach already started with the *privileged* flag for aspects, that allows access beyond the normal scope (also cf. Section 4.2.6). In Figure 4.10 we show a potential syntax with a *context* keyword to refactor the example from Figure 3.7a (p. 40). With exposed local variables all workarounds described above would not be necessary and extensions would be much easier to develop, and probably even more maintainable than the workarounds.

4.2.5 Advise Objects Problem

A specific problem we observed some times during the refactoring is that only certain objects should be advised, not all instanced of a class. AspectJ does not provide direct support for this.

An simple example is extension from the MEMORYBUDGET feature for the ATOMICTRANSACTIONS feature. The former feature observes how many read and write locks are held by a transaction, so each time a lock is acquired or released the memory size is updated. The memory budget is informed every time when a lock is added or removed from one of the lists. However, the extensions for read locks and write locks are different because the lock types have a different memory overhead. In Berkeley DB's implementation this results in scattered code because every single access is extended individually.

AspectJ does not support to extend all call join points for methods of a specific object, but only for a whole class. It is only possible to use the *withincode* pointcut to narrow down the extended join points to certain methods, but in our example this does not work because there are cases where both lists are used in the same method. Alternatively, also the pointcut *if* can be used for a conditional extension.

However, the solution is simple and does not require AspectJ. Berkeley DB could just implement two different memory observing lists that update the memory changes themselves. Now refactoring MEMORYBUDGET code only requires to remove the target code from the memory observing list classes. Why this approach was not chosen in the first place is questionable, but as the code is repeated only few times the developers of Berkeley DB

might not even recognize this as a problem.

When we tried to implement such a solution with AspectJ, we observed a problem with pointcut semantics, which we already discussed in Kästner et al. (2006). Certain semantic rules—that are not described formally but only observable with experiments—seem strange and prevent simple solutions. We used an empty class *ReadLocksSet* inherited from the standard Java class *HashSet* as shown in Figure 4.11. From the four different extensions only 4.11d and 4.11e are possible with the observed AspectJ semantics, while the more intuitive solutions 4.11b and 4.11c have no effect, surprisingly for most developers.

The solution from Figure 4.11b, using a call pointcut, does not work because the target object is only defined as a *Set*, not as a *ReadLocksSet*. Even though the effect during object-oriented development is the same, AspectJ distinguishes between different target classes. The solution from Figure 4.11c, using an execution pointcut, does not work because extension pointcuts can only extend methods that are literally defined in the target class, but does not work on inherited methods[3]. To use this extensions it would be necessary to define every extended method in *ReadLocksSet* in the base code with empty methods that only call the parent's implementation. The extension shown in Figure 4.11d with a ITMD appears promising but has a problem when different features want to extend the same method, because it can only be introduced once. Finally, the solution shown in Figure 4.11e was used during our refactorings but is a conditional extension that is only evaluated at runtime, equivalent to the *instanceof* implementation shown in Figure 4.1b (p. 55).

As an alternative it is often possible to defer the introduction of the specialized class to the feature. In our example it would also be possible to create the *ReadLocksSet* subclass inside a feature and use an around advice to create an instance of this class, instead of the general *HashSet* class, e.g., *'HashSet around() : execution(HashSet.new()) && withincode(...) { return new ReadLocksSet(); }'*. This is similar to mixin based inheritance (Bracha and Cook, 1990), however like the extension above using ITMDs, this approach does not scale when multiple features extend one object.

All in all, extensions to objects instead of classes can usually be resolved with subclassing, without AspectJ. When refactoring these extensions into features it is necessary to observe the undocumented, sometimes unexpected semantics of call and execution pointcuts shown above. These semantics are also observed when emulating mixin inheritance with AspectJ.

[3]As explained in Kästner et al. (2006), AspectJ does not have a formal description of its semantics. Thus, we have to rely on empirically derived semantics from an AspectJ compiler. However, even between different compiler versions *(ajc version 1.2 vs. 1.5)* and different vendors *(ajc vs. abc)*, we found minor variations in the semantics. For example, the call pointcut of Figure 4.11b also requires literally defined methods like the execution pointcut in ajc version 1.2, but does not in ajc version 1.5. Because of the absence of formal semantics we cannot distinguish between intended behavior and compiler bugs. For our analysis we refer to the semantics that can be experimentally determined from the ajc compiler version 1.5, which we used for our refactorings.

```
1  public class ReadLocksSet extends HashSet {}
2
3  public class Txn {
4    private Set readLocks = new ReadLocksSet();
5
6    void addLock(Lock lock, LockType type, ...) {
7      ...
8      if (type.isWriteLock()) {
9        ...
10       readLocks.remove(lock);
11     } else {
12       readLocks.add(lock);
13     }
14   }
15   ...
16 }
```

(a) Base code.

```
1  public aspect TransactionMemory {
2    after(Txn txn) : call(boolean
         ReadLocksSet.add(Object)) &&
         this(txn) {
3      txn.updateMemoryUsage(
           READ_LOCK_OVERHEAD);
4    }
5  }
```

(b) Extensions with call pointcut.

```
1  public aspect TransactionMemory {
2    after(Txn txn) : execution(boolean
         ReadLocksSet.add(Object)) &&
         this(txn) {
3      txn.updateMemoryUsage(
           READ_LOCK_OVERHEAD);
4    }
5  }
```

(c) Extensions with execution pointcut.

```
1  public aspect TransactionMemory {
2    public boolean ReadLocksSet.add(Object
         o) {
3      txn.updateMemoryUsage(
           READ_LOCK_OVERHEAD);
4      super.add(o);
5    }
6  }
```

(d) Extension with a ITMD.

```
1  public aspect TransactionMemory {
2    after(Txn txn) :
         call(boolean Set.add(Object)) &&
         target(ReadLocksSet) && this(txn) {
3      txn.updateMemoryUsage(
           READ_LOCK_OVERHEAD);
4    }
5  }
```

(e) Conditional extension with target pointcut.

Figure 4.11: Different Approaches to Advise an Object.

4.2.6 Scope Problem

A last problem that frequently occurred during our refactorings was the scope of classes, methods, and member variables. Advice is not executed in the same scope as the method it extends, but in the scope of the aspect, similar to a method in a different class. Hence, it cannot access private methods or member variables of the class it extends.

Berkeley DB uses the Java scope model very carefully to only publish methods publicly that can be called by the user. Also protected, private, and default[4] scope are selected carefully and appropriately. The selected choice often does not fit to the requirements of aspects that need public methods.

Developers of AspectJ anticipated this problem and introduced the *privileged* flag for aspects. Privileged aspects can access methods and member variables that would otherwise be hidden by the Java scope system (AspectJ-Team, 2003; Laddad, 2003, p. 139ff). This ability to bypass Java's scope system is discussed critically for violating the object-oriented encapsulation principle (Chiba and Ishikawa, 2005; Aldrich, 2004). Therefore, it is usually recommended to avoid privileged aspects if possible. For example Laddad (2003) suggests to use it only when extending implementation details, because otherwise the aspect could depend on interface methods and be less fragile to implementation changes of the class. Unfortunately, in our refactoring project we almost only refactor and extend implementation details, therefore we constantly need privileged aspects. Of all 151 Aspects used for our refactorings 127 were declared privileged.

Where the default scope was used in Berkeley DB the aspect had to be declared in the same package. This prevents extensions from one aspect to methods in different classes in different packages, which was used in multiple cases. Instead one aspect must be created per package, which breaks the locality and encapsulation of semantically similar extension, which was just achieved with this aspect.

Even though aspects can bypass the scope system when declared privileged, we still had to change scope modifiers frequently for three reasons:

1. Declaring an aspect privileged does not help when a class with private or default scope must be imported from a different package. The import statement still checks the scope constraints and forces us to make the target class public. The same problem exists with inner classes that typically not declared public and therefore not in the scope of the aspect.

2. In some cases we reintroduced large pieces of code as subclasses, emulating mixin inheritance. For these classes we again had to change scope modifiers because the *privileged* modifier is not available for classes. Hence, aspects and class extensions are treated differently when using AspectJ to implement features.

3. For no apparent reason, AspectJ does not allow protected ITMD. When we reintroduced protected methods or fields that can be overwritten by subclasses we had to

[4] Also known as *package* or *friendly visibility*. Only classes or aspects from the same Java package can access the method.

declare them with public or when possible default scope. Not even abstract methods can be introduced with a protected scope.

Finding a solution for this problem is difficult because it is a fundamental problem of the way AspectJ weaves extensions from a third person perspective. Changes like introducing the *privileged* flag for classes, allowing protected ITMDs, or changing the import mechanism only fix the symptoms but do not improve the basic problem of extending source code from a different perspective. With a different approach like preprocessor statements, AHEAD's *jampack* composition (cf. Batory et al., 2004), or our *ColoredIDE* (cf. Section 9.2) this problem can be avoided.

To sum up, when decomposing an application with AspectJ scope problems are very likely. Some can be worked around with the *privileged* modifier while others require changes to the base code to declare the classes, methods or member variables public. After all these changes do not prevent the refactoring into features, just have unpleasant effects and can destroy a thoughtfully designed scope concept.

4.3 Summary

The analysis of the AspectJ languages constructs shows that for the refactoring of Berkeley DB most language features are not needed. Only static introductions, simple method refinements, and statement extensions were needed, while more advanced and specific constructs like call extensions, conditional crosscuts, or homogeneous crosscuts are used only in rare cases. Table 4.2 sums up the our collected statistics and shows the number of occurrences for each type as well as the proportion of all extensions.

Metric	NOO	PAE
Static extensions	640	56.9
Interface Introductions	4	0.4
Class Introductions	58	5.2
Method Introductions	365	32.5
Member Variable Introductions	213	19.0
Dynamic Extensions	484	43.1
Method refinements	214	19.0
Statement extension emulations	121	10.8
Extensions with hook methods	164	14.6
Conditional extensions	5	0.4

NOO: Number of occurrences; PAE: Percentage of all extensions

Table 4.2: Extensions used in Refactoring Berkeley DB.

These statistics are a hint which features are needed for a language that could be used to implement features. Possibilities to express statement extensions are more important than

highly complex constructs to extend the control flow or to quantify over join points for homogeneous extensions.

Furthermore, the existing statements often were not sufficient. We reached limits of the AspectJ language during our refactorings multiple times. Statement extensions are missing from AspectJ and can only be emulated in some cases with more complex constructs. The access to local variables is not intended and only possible with workarounds that increase the complexity of the source code, even though it is often needed. Furthermore, the commonly needed introduction of new parameters or exceptions is not possible, and AspectJ has problems with advising objects and the scope of classes, methods, and member variables.

Even though we suggested problems to every of these problems, our results raise the serious question if, even with all fixes, AspectJ can be considered as a reasonable language to implement features.

Chapter 5

The Feature Optionality Problem

During the refactoring of Berkeley DB, we observed frequently the Feature Optionality Problem. The Feature Optionality Problem is a well known problem when dealing with optional features in a Software Product Line (Prehofer, 1997; Liu et al., 2005, 2006). It occurs when multiple optional features interact with each other, e.g., feature A refers to feature B or feature A extends feature B. When the interaction code is included in feature A then this feature depends on the presence of feature B. Feature A can no longer be composed individually but only together with all features it refers to or extends. As stated in Liu et al. (2005) this *"is an undesirable effect that undermines feature reusability, as feature optionality can no longer be achieved"*.

In Figure 5.1 we show a simplified example of this problem. The *Environment* class from the base code of Berkeley DB is extended by two aspects from the features MEMORYBUDGET and STATISTICS. The latter collects statistics from the base code (Lines 14–15) and from the MEMORYBUDGET feature (Lines 16–18). The STATISTICS feature cannot be woven if the MEMORYBUDGET feature is not present, because it requires the member variables *maxMemory* and *usedMemory* introduced by that feature.

During our refactorings we observed many such dependencies that cause the Feature Optionality Problem. All in all, we found 43 implementation dependencies in our feature model, as depicted in Table 3.3 (p. 51). Just the original statistics feature requires 7 more features, which again interact with other features. Thus, the feature STATISTICS requires—directly or indirectly—a total of 15 other features[1]. Similarly, other features like LATCHES or MEMORYBUDGET also have many direct and indirect dependencies. There is even a bidirectional dependency, i.e., MEMORYBUDGET extends LATCHES while LATCHES references MEMORYBUDGET, therefore neither can be included without the other. With all these implementation dependencies Berkeley DB cannot be considered decomposed, because virtually all refactored core features are required in almost all possible configurations. Even if a user wants to configure a small version with only few features, e.g., statistics but not transactions, he is forced to include many other features.

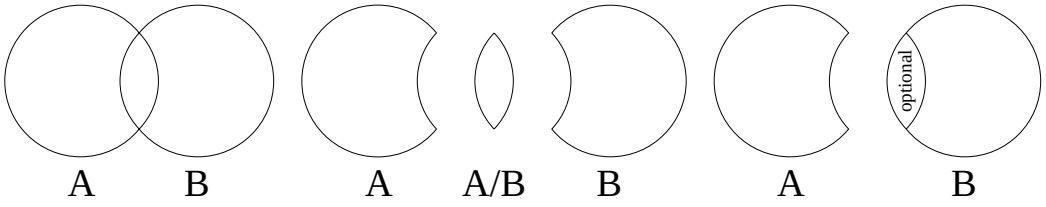
[1] STATISTICS implies ATOMICTRANSACTIONS, CHECKPOINTER, DBVERIFIER, DELETEDBOPERATION, EVICTOR, FILEHANDLECACHE, FSYNC, INCOMPRESSOR, LATCHES, LEAKCHECKING, LOCKING, MEMORYBUDGET, TREEVISITOR, and TRUNCATEDBOPERATION.

```
1  class Environment {
2    void openDb(String name) { ... }
3    byte[] readData() { ... }
4  }
```

```
5  aspect MemoryBudget {
6    long Environment.maxMemory;
7    long Environment.usedMemory=0;
8    after(Environment env) : execution(void openDb(String)) && this(env) {
9      env.usedMemory++;
10   }
11   ...
12 }
```

```
13 aspect Statistics {
14   after() : execution(byte[] readData()) { ... }
15   int Environment.getBufferHitRatio() { ... }
16   long Environment.getFreeMemory() {
17     return maxMemory-usedMemory;
18   }
19 }
```

Figure 5.1: Feature Optionality Example.



(a) Two interacting features. (b) Derivative Feature Approach. (c) Optional Weaving Approach.

Figure 5.2: Resolving Feature Interactions

We found two different approaches to tackle this problem, the Derivative Feature Approach and the Optional Weaving Approach.

5.1 Derivative Feature Approach

A basic idea to break up these dependencies is to encapsulate the code that implements interactions in separate modules (Prehofer, 1997). In our infrastructure that requires to create a new feature for every interaction. Because these features for interactions are derived from the features they connect, we call them *derivative features*. We use the naming convention A/B for a derivative feature that is derived from the features A and B. In Figure 5.2b we show a visual representation of this approach: the overlapping code, i.e., code that implements interactions, is moved to a separate derivative feature.

In the above example from Figure 5.1, we would create one feature that contains all changes made by MEMORYBUDGET to the base code, a second feature that contains all

```
1 aspect Statistics {
2   after() : execution(byte[] readData()) { ... }
3   int Environment.getBufferHitRatio() { ... }
4 }
```

```
5 aspect StatisticsMemoryBudgetDerivative {
6   long Environment.getFreeMemory() {
7     return maxMemory-usedMemory;
8   }
9 }
```

Figure 5.3: Derivative for the Example.

changes made by STATISTICS to the base code, and a derivative feature STATISTICS/MEMORYBUDGET that contains all changes that are applied only when both features are included in a target configuration. Thus, the second feature implements only the changes made to the base code that are independent of feature MEMORYBUDGET, in this case the buffer hit ratio calculations (Lines 14–15). In contrast, the derivative feature implements the STATISTICS's extension for MEMORYBUDGET, i.e., the *getFreeMemory* method that depends on the member variables introduced by MEMORYBUDGET (Lines 16–17). In Figure 5.3 we show the resulting new *Statistics* aspect and the new derivative feature.

This approach was formalized with a mathematical model by Liu et al. (2006, 2005). It enables to resolve dependencies into derivatives features completely and to reason algebraically about their composition. The model is more detailed than our derivative features above, because it separates between code introduction and code transformation and between different types of derivatives that need to be applied when two features are selected. Following this formal approach strictly, a very fine granularity of derivatives is possible, which also results in a very high number of derivative modules. For the practical decomposition the 'informal' Derivative Feature Approach shown above is sufficient, though it still can be mapped to the mathematical model if required.

We used the Derivative Feature Approach exemplary on the STATISTICS feature in Berkeley DB. We refactored all code to collect statistics into derivative features like STATISTICS/DATABASE or STATISTICS/TRANSACTIONS, because it interacts with other features and thus depends on them. Only a basic infrastructure for statistic calls remains in the final STATISTICS feature. So, if the optional ATOMICTRANSACTIONS feature is not included, the feature STATISTICS/TRANSACTIONS cannot be included either, but other parts of the statistic system are not affected and can be composed independently. Overall, we created 12 derivative features for the original STATISTICS feature, to resolve all dependencies[2].

The resulting feature model of the statistics system is shown in Figure 5.4. The figure shows the 12 individual derivative features that each have one or two specific semantic

[2]To be precise, some of these derivative features currently do not have dependencies, because their target features have not been refactored, yet. A refactoring is anticipated though. For example, the STATISTICS/LOG feature has no dependencies yet, because the required code of the persistence layer is still in the base code. Once the persistence layer is also refactored there will be a dependency to this feature.

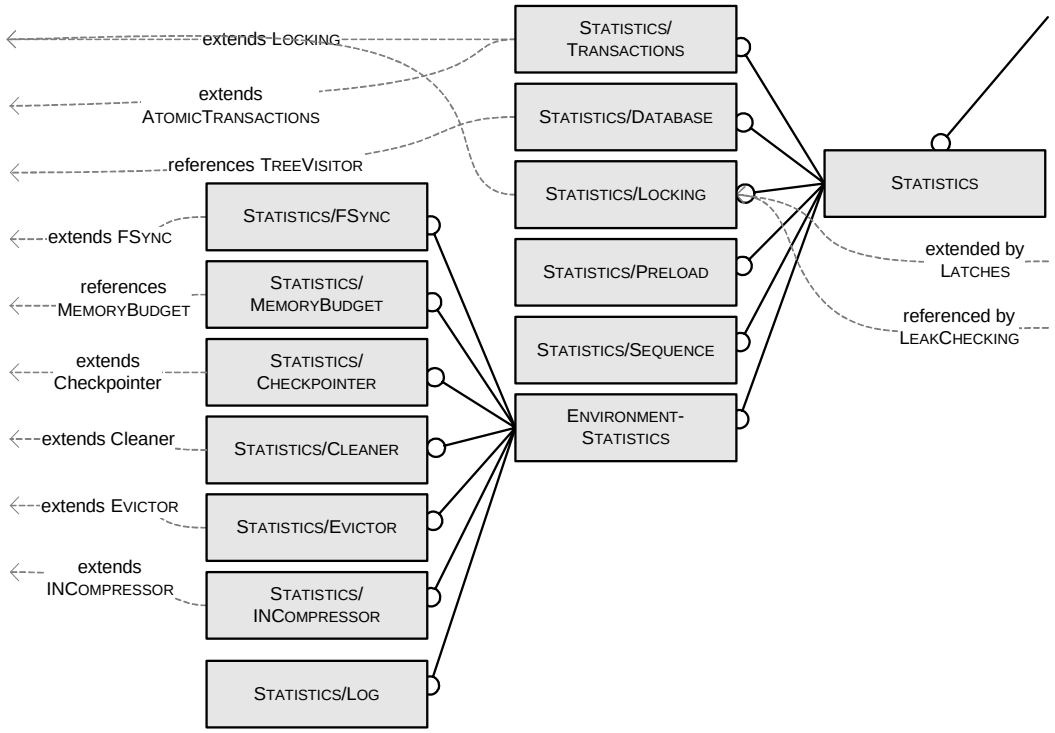


Figure 5.4: Resolved Statistics Subsystem Feature Diagram.

dependencies[3], instead of the original STATISTICS feature in Figure 3.8 (p. 47) with its many implementation dependencies.

After applying the Derivative Feature Approach, the statistics system in Berkeley DB can be used independently of the presence of other features.

5.1.1 Semantic vs. Implementation Dependencies

Already in Section 3.5 we have introduced the difference between semantic and implementation dependencies. Semantic dependencies are dependencies where one feature requires the other feature because it does not make sense otherwise. For example the feature CRITICALEVICTION calls the *Evictor* before critical operations to ensure enough available memory . When the EVICTOR feature that contains the code to free memory is not included, CRITICALEVICTION cannot perform anything. On the other hand implementation dependencies occur because the implementation of one feature depends on another feature, while a different implementation could work without this dependency. For example, the feature STATISTICS also requires EVICTOR, however most statistics can also be collected without the EVICTOR feature.

[3]Note, the ENVIRONMENTSTATISTICS feature is only introduced for technical reasons to provide a common infrastructure for some of the derivative features.

The derivative features approach splits features into smaller features, e.g., STATISTICS in a newer, smaller STATISTICS feature and the STATISTICS/EVICTOR derivative feature. This way, the implementation dependencies are resolved and replaced with semantic dependencies. The STATISTICS feature not longer depends on EVICTOR, but STATISTICS/EVICTOR does. However, the dependency between STATISTICS/EVICTOR and EVICTOR is a semantic dependency. To collect statistics about the Evictor does not make sense when the Evictor is not present in the system.

This way, it is possible to replace all implementation dependencies with semantic dependencies of smaller features. A feature model without any implementation dependencies is possible, which allows a much more flexible composition.

5.1.2 Implementing Derivatives

In Berkeley DB, we implemented derivative features like normal features with AspectJ. Hence, a derivative feature can introduce code or extend the base code or required features, i.e., the features is is derived from. For example, the STATISTICS/EVICTOR derivative feature implements the interactions between the two features STATISTICS and EVICTOR. It extends the *Evictor* class and code that the EVICTOR feature introduces into the B^+-tree. Furthermore, the derivative feature extends the statistics infrastructure introduced by the STATISTICS feature.

Technically, such derivative modules are possible because classes introduced by other features are available in the build path due to the infrastructure we imposed on our project (cf. Section 3.1). Furthermore, AspectJ allows to quantify over join points introduced by other aspects, e.g., methods introduced using ITMDs. Our implementations of derivative features generally do not extend aspects but the code introduced by aspects, because extending advice is hard in AspectJ. The extension of advice from other aspects is possible only implicitly by advising the same join points and using a *declare precedence* statement. In AspectJ, aspects provide join points, too, therefore it is possible to advise member functions—but not individual advice statements[4]—of aspects. Nevertheless, during the refactoring, we hardly used member methods of aspects, but usually introduced code into classes from the base code[5]. Thus, in our experience, extensions to code from other features are almost always pieces of advice extending methods introduced by these features.

A different implementation technique we considered was *aspect inheritance*. Even though inheriting from other aspects is technically possible, it could not be used for derivative features because of two problems. The first problem is that advice cannot be refined. Aspect inheritance can only be used (1) to extend the aspect's member methods, (2) to use abstract pointcuts that are implemented in a child aspect, or (3) to reuse pointcuts from parent aspects (AspectJ-Team, 2003; Apel et al., 2006b). This limits the usability of aspect

[4] In theory it is also possible to use the *adviceexecution* pointcut designator to advise advice statements. However, because advice is unnamed and not a first class entity, it is not possible to extend a specific piece of advice, but only all advice statement in an aspect at once. This makes it unusable for practical refactoring.

[5] In all 38 features with overall 151 aspects we only used 6 methods inside aspects compared to 365 ITMDs to introduce methods.

inheritance for derivative features drastically. The second problem is that aspects cannot inherit from concrete aspects, i.e., aspects that are not declared abstract (AspectJ-Team, 2003; Laddad, 2003; Apel et al., 2006b). When dealing with optional features this prevents the use of aspect inheritance, because aspects are extended only when a certain feature is selected. Which feature provides the last and thus concrete aspect in the hierarchy is not known at design time, but only at compilation after a configuration is selected. These problems and a possible solution based on mixin based aspect inheritance are discussed in detail by Apel et al. (2006b). The ability to use aspect inheritance for derivative features to refine advice would be useful in some (currently rare) cases where in our current solution both aspects extend the same join point. On the other hand, the ability to refine pointcuts is not needed in Berkeley DB because hardly any homogeneous pointcuts were used that could be modified with such refinements.

The AspectJ limitations discussed in Section 4.2, like the Parameter Introduction and Passing Problem, the Exception Introduction Problem, or the Local Variables Access Problem, also affects derivative features. For example, in some cases a new parameter or a new exception is the only thing introduced by the derivative features, e.g., the derivative feature for ATOMICTRANSACTIONS and TRUNCATEDBOPERATION only introduces a *transaction* parameter into the interface method. This emphasizes the urgency to solve these problems before a serious refactoring of Berkeley DB is reasonable.

Nevertheless, the major problem of implementing derivative features was a practical one: insufficient tool support. For example, AJDT fails to correctly show advice on methods introduced by ITMDs and even mark calls to methods introduced by other aspects as an error. Such problems make development harder and more error-prone because the developer must rely on runtime tests to determine if the features were correctly refactored. Nevertheless, this is just a tool problem, not a general problem of AspectJ.

To summarize, it is possible to implement derivative features in AspectJ, although the language requires some workarounds. The language limitations discussed earlier also occur in derivatives features, and insufficient tool support is a major practical problem.

5.1.3 Scale

When the number of features in a system increases, also the number of derivative features grows, thus raising the overall complexity of the system.

Higher Order Derivatives. The concept of derivative features becomes more complex when more than two features interact. In this case the developer must create *higher order derivatives*, that encapsulate their interaction. For example, the second order derivative STATISTICS/MEMORYBUDGET/EVICTOR contains interaction code that is only included when the three features STATISTICS, MEMORYBUDGET, and EVICTOR are selected in a configuration. As shown by Liu et al. (2005) this concept scales for arbitrary high ordered derivatives, but the number of possible derivatives grows exponentially with the number of features in the project.

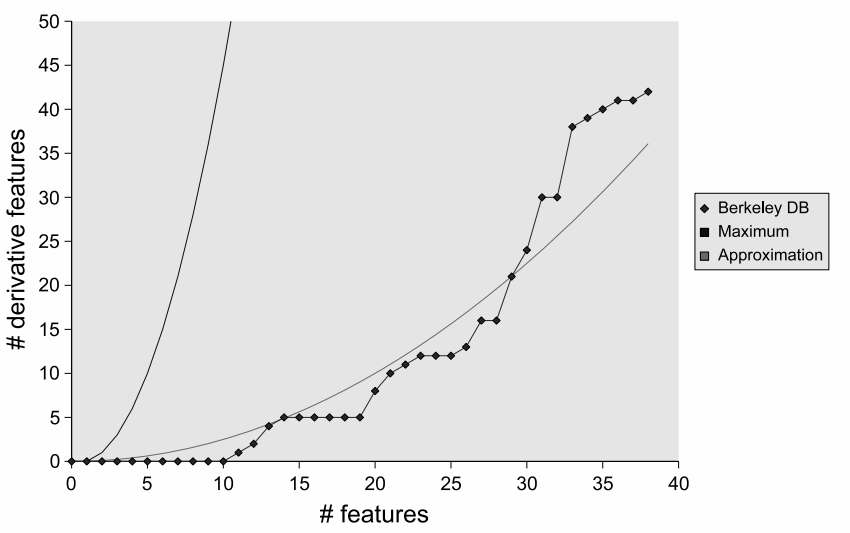


Figure 5.5: Scale of Derivative Features.

However, in Berkeley DB we could express all interactions we found with first order derivatives, i.e., all interactions can be broken down to mutual interactions between two features. This corresponds to observations made by Liu et al. (2005) and suggests that the *maximum number* of derivative features grows quadratically with the number of features[6].

Number of Derivative Features. The number of derivative features affects how a system with many features is manageable or maintainable. The above considerations show that theoretically any pair of features can have a derivative. So, in a small system with 5 features 10 derivative features are possible, which seems manageable. In Berkeley DB with currently 38 features, 703 derivatives features are possible. And in a system refactored into 250 fine grained features already 31125 features are possible, which appears unmanageable. This shows that it is important to predict how many derivative features are to be expected. If the actual number of derivative features is close to the maximum only small projects with few features are manageable at all.

To approximate the effect of scale further, we have observed the number of derivative features during our refactorings. In Figure 5.5 we show how many derivative features were required for each number of refactored features. This Figure shows that the first features refactored do not interact at all, and the following features require only few derivatives. However, the more features there are, the more derivatives occur with refactoring a further feature. The most derivatives were added by the 33rd feature (MEMORYBUDGET) which interacts with 8 previous features.

This observation shows that there are much less derivative features than possible. Most refactored features do not interact, or interact only with few other features. However, the

[6]To be precise, $\binom{n}{2} = \frac{n^2-n}{2}$ derivative features are possible for n features. These are all pairs while the order does not matter.

observation also shows that the number of derivatives grows with the number of features and the growth rate increases with the number of features. Overall, this again points to a quadratic growth, but the growth is much slower than the maximum number. A quadratic approximation to the observed derivative feature growth is $0.025 \cdot n^2$ derivative features for n features as shown as 'Approximation' in Figure 5.5.

To predict further growth empirically our sample size is too small. Therefore, we tried to predict further growth with a dimension model that groups the features into four dimensions and makes assumptions about the interactions of these dimensions. Inside one dimension, like *database operations* or *memory and caching*, only few semantic dependencies occur, while features from a dimension typically interact with all other dimensions. We quantified this model and created simple equations to calculate the number of derivative features approximately. This model confirms our theory that the number of derivative features increases quadratically but slower than the maximum number. The full dimension model is explained in Appendix A.

Both empirical observations and dimension model point to a quadratic grows similar to $k \cdot n^2$ with $k \ll 1$. Such a growth inflicts various problems on scaling feature models. When a feature model grows there will always be a point at which the number of derivative features becomes very high. For example, in a project with 250 features and a growth of $0.025 \cdot n^2$ we would still need 1562 derivative features. This shows that the approach can either be used for small to medium sized feature models only, or there must be some means to hide or inline derivative feature to reduce complexity.

Hiding Complexity. The derivative features are solely necessary for technical reasons. The user who configures his version of Berkeley DB does not need them[7]. Thus, hiding them might decrease the perceived complexity.

In fact, the derivative features can be selected automatically, thereby hiding a certain level of complexity from the end-user. With a tool it is possible to select the derivative features based on the user's selection of features in Berkeley DB. In Figure 5.6 we show one possibility to implement such an automated selection with a *guidsl* grammar. The user only selects the original features and the derivative features are selected automatically with the conditions in Lines 11–18. Unfortunately the *guidsl* tool cannot hide the derivatives completely, yet. Alternatively, also the concept of origami can be used to hide derivatives using the AHEAD infrastructure (Batory et al., 2003), or new tools might be developed.

Such tools do not remove the complexity. A developer still must create derivative features and maintain the feature model and the grammar for automated selection. However, it is at least possible to hide the additional complexity caused by the Feature Optionality Problem from the end-user.

[7]While a user might still chose which statistics he wants to collect, other derivatives like the one that implements the interaction between the MEMORYBUDGET and the EVICTOR features are purely technical.

```
1  BerkeleyDb : Derivatives* [STATISTICS] [MEMORYBUDGET] [EVICTOR] [ATOMICTRANSACTIONS];
2
3  Derivatives
4      : STATISTICS/TRANSACTIONS
5      | STATISTICS/EVICTOR
6      | STATISTICS/MEMORYBUDGET
7      | MEMORYBUDGET/EVICTOR;
8  ...
9  %%
10 //automatic derivative selection
11 STATISTICS and EVICTOR implies STATISTICS/EVICTOR;
12 STATISTICS and MEMORYBUDGET implies STATISTICS/MEMORYBUDGET;
13 STATISTICS and ATOMICTRANSACTIONS implies STATISTICS/TRANSACTIONS;
14 MEMORYBUDGET and EVICTOR implies MEMORYBUDGET/EVICTOR;
15 not STATISTICS or not EVICTOR implies not STATISTICS/EVICTOR;
16 not STATISTICS or not MEMORYBUDGET implies not STATISTICS/MEMORYBUDGET;
17 not STATISTICS or not ATOMICTRANSACTIONS implies not STATISTICS/TRANSACTIONS;
18 not MEMORYBUDGET or not EVICTOR implies not MEMORYBUDGET/EVICTOR;
19 //semantic dependencies
20 ...
```

Figure 5.6: Automatic Derivative Selection using a *guidsl* Grammar.

5.2 Optional Weaving Approach

A different approach to solve the Feature Optionality Problem stems from the fact that advice which does not match any join points neither is woven nor does it create a compilation error. Leich et al. (2005b) suggest to implement optional interactions in features themselves, but with language constructs as advice statements that are ignored when the second feature is not selected. We call this approach the *Optional Weaving Approach*. In Figure 5.2c we show a visual representation: the interaction code still remains in one feature, but is now optional.

The implementation of our previous example with this approach is shown in Figure 5.7: The aspect *Statistics* no longer references the *MemoryBudget* aspect directly. Instead, it advises the *incUsedMemory* method introduced by *MemoryBudget*. When the *MemoryBudget* aspect is not included for compilation this advice statement is not woven, and the variable *statsUsedMemory* is therefore always zero. With this approach we are able to resolve the implementation dependency and to compose both features individually.

The advantage of the Optional Weaving Approach compared to the Derivative Feature Approach is that it is not necessary to create derivative features to resolve dependencies. Dependencies can be implemented as optional extensions inside the original feature definition and can be maintained locally, similar to the way it was maintained in the original code. This means that the Optional Weaving Approach does not have problem with scale and does not need any tool to hide complexity. The interactions are woven automatically by the AspectJ compiler when the depending feature is selected.

Although this approach has these advantages, it has several disadvantages, which make it unusable for practical refactoring with the current AspectJ language:

- First, it is not possible to reference optional classes, methods or member variables in an optional advice statement. For example, the advice statement in Figure 5.7 cannot

```
20 class Environment {
21   void openDb(String name) { ... }
22   byte[] readData() { ... }
23 }
```

```
24 aspect MemoryBudget {
25   long Environment.maxMemory;
26   long Environment.usedMemory=0;
27   void Environment.incUsedMemory(){
28     usedMemory++;
29   }
30   after(Environment env) : execution(void openDb(String)) && this(env) {
31     env.incUsedMemory();
32   }
33   ...
34 }
```

```
35 aspect Statistics {
36   after() : execution(byte[] readData()) { ... }
37   int Environment.getBufferHitRatio() { ... }
38
39   long Environment.statsUsedMemory;
40   after(Environment env) : execution(void Environment.incUsedMemory()) && this(env) {
41     env.statsUsedMemory++;
42   }
43   long Environment.getUsedMemory() {
44     return statsUsedMemory;
45   }
46 }
```

Figure 5.7: Optional Advice Example.

access the member variables *maxMemory* or *usedMemory* which were introduced by the *MemoryBudget* aspect. Therefore, *Statistics* mirrors the logic and values, i.e., it monitors the memory usage in its own variable instead of using the existing one from *MemoryBudget*. Thus, the optional advice approach introduces code replication.

- Second, optional weaving is possible only for advice statements, not for ITMDs. This means that the target for ITMDs must exist and cannot be optional. Additionally, all extensions that belong to this interaction, but are not implemented with optional advice are woven anyway. In our example the member variable *statsUsedMemory* and the method *getUsedMemory* are introduced in any case, even when the advice statement is not woven. As a consequence, *getUsedMemory* is always present but returns an empty value when compiled without the *MemoryBudget* aspect. Thus, the Optional Weaving Approach can introduce unnecessary variables and methods, that have no runtime semantics without the interactions.

- Finally, the scope problems shown above also hinder advising methods in optional classes. To define the advice the target class must be in the scope of the aspect, i.e., imported with the Java *import* statement. If the class is excluded from compilation this *import* statement will issue an error. Similarly, optional classes can never be referenced in any advice declaration, e.g., it cannot be used as a parameter and captured with a *this* or *args* pointcut.

In general, avoiding the complexity of derivative features is a promising aim, however language restrictions make it hard or even impossible to implement optional interactions with this approach. With the current AspectJ language, this approach leads to code replication and the weaving of unnecessary code without runtime semantics.

A possible solution would be to define groups of statements—advice and ITMDs—as an optional extension, where the whole group has transactional semantics, i.e., it is either woven with all advice and introductions, or no statement is woven. When one advice statement in this group cannot be woven, all other group members are also excluded from compilation. Furthermore, references to unavailable classes in not woven code must not produce a compilation error, so that access to classes, methods, and member variables introduced by other features is possible inside an optional extension group. Such optional extension groups are similar to derivative features, but they are defined inside a feature and are compiled automatically. Hence, they do not create huge and complex feature models. A simple way to implement such groups as an extension to AspectJ is to tag all members of this group with an annotation and to extend the AspectJ compiler to weave all annotated members with transactional semantics. However, as long as such language improvements are not available, the Optional Weaving Approach cannot be used as a solution for the Feature Optionality Problem.

5.3 Summary

While refactoring Berkeley DB, we observed the Feature Optionality Problem, that hinders independent composition of optional features. We tried to solve the problem with the *Derivative Feature Approach*, based on existing research on derivatives and origami (Prehofer, 1997; Liu et al., 2006; Batory et al., 2003). First refactorings were promising, but we observed limitations in the AspectJ language again, as well as an increasing complexity of the feature model. The number of derivative features is expected to grow quadratically with the number of features in the project, which hinders scaling feature models.

On the other hand, the alternative *Optional Weaving Approach* (Leich et al., 2005b) tackles the problem with extensions inside the feature that are only woven when the depending feature is included. This avoids the complexity of the Derivative Feature Approach, however it is limited by the AspectJ language that does not support optional weaving well. Therefore this approach cannot be used practically.

These results show that the Feature Optionality Problem is a major problem, especially for scaling feature modeling to models with many features. For our refactorings so far, the Derivative Feature Approach is manageable, but as soon as the number of features increases, e.g., as soon as we refactor further database operations, this becomes too complex for practical use. The optional weaving problem could not be tested, due to insufficient language support. The Feature Optionality Problem is an important field of research, and further work that goes beyond theoretical models is needed. A solution to solve the Feature Optionality Problem must be stable, less complex, and convenient for the developer and the end-user.

Chapter 6

Feature Order

A premise in FOP is that the order in which features can be composed is fixed and features are applied sequentially as functions. In contrast, AspectJ applies all aspects 'at once', i.e., aspects can advise the full source code including changes made by other aspects. When two or more advice statements extend the same join point, the order in which they are applied is calculated by precedence rules or the order is simply undefined (Kiczales et al., 2001; Lopez-Herrejon et al., 2006).

There are several reasons why a fixed weaving order should be considered also for AOP:

1. **Shared Join Points.** In cases where a join point is extended multiple times the order in which extensions are applied can make a difference. For example, if a synchronization and a logging extension is applied to an interface method the order decides whether the logging code is synchronized or not. An explicit weaving order can pinpoint the precedence.

2. **Accidental Weaving.** As argued by Lopez-Herrejon et al. (2006) and McEachen and Alexander (2005), the 'at once' weaving of all advice is potentially dangerous because pattern expressions can match code that was not intended. This can lead to serious consequences like deadlocks. To avoid these problems, Lopez-Herrejon et al. (2006) have suggested to weave aspects like functions in a fixed order and thus to bound aspect quantification.

3. **Aspect Extensions.** Similarly, Forster and Steimann (2006) have shown that the global 'at once' weaving approach can produce logical problems, resulting in infinite loops in AspectJ. They suggest an ordered weaving similar to the one proposed by Lopez-Herrejon et al. (2006), but with different language constructs.

4. **Reduced Complexity.** The 'at once' weaving inflicts that each aspect can interact with any other aspect, for n aspects there are n^2 possible interactions. Using the bounded quantification approach every feature can interact only with previous features, not with later ones. The number of interactions is thus decreased by 50 % (Apel and Liu, 2006).

5. **Integration of AOP and FOP.** When aspects are integrated in an FOP approach, e.g., AML (Apel et al., 2006c), a bounded quantification is crucial to get a consistent behavior of FOP artifacts like mixins (Batory et al., 2004; Bracha and Cook, 1990) and aspects. In Apel et al. (2006b) and Kästner et al. (2006), we have already proposed and partly implemented a compiler to integrate aspects to AHEAD and to bounded aspect quantification. This compiler needs an ordered list of features and compiles them sequentially.

In the remainder of this section we discuss the first three points, how AspectJ deals with shared join points, if and how accidental weaving occurred, and if problems based on aspect extensions were relevant. The integration of AOP and FOP is outside the scope of this paper and discussed elsewhere (e.g., Apel et al., 2005, 2006c,b; Mezini and Ostermann, 2004).

6.1 Observations on Shared Join Points

During our refactorings, almost all extensions were done in disjoint points in Berkeley DB. We found that only 28 of overall 528 extended join points (5 %) were extended with more than one advice statement. These join points are called *Shared Join Points (SJPs)*.

Furthermore, we made the following observations about SJPs in Berkeley DB:

- In 7 cases the affected SJP is a constructor or an *init* method, where different features initialize variables in their extensions. In all these cases the weaving order does not matter.

- In 6 cases a SJP is extended twice but with different advice that automatically specifies the feature order: One feature uses before advice and the other feature uses after advice. Even though both advice statements extend the same join point the order is defined already by the used advice independent of aspect precedence.

- The other SJPs can be classified as following: 2 join points are extended seven times by different statistic features. Every static feature extends the *clear* and *toString* method to reset and print statistics. Especially for the latter the order matters. 4 join points are extended four times by each daemon thread to initialize, run, pre-shutdown, and shutdown these daemon threads in the correct order. The remaining 9 SJPs are all extended twice only and involve the features STATISTICS, LATCHES, and MEMORYBUDGET. In many of these cases it is not obvious whether the order is important or not, therefore we enforced the original order in every case.

- We found 12 additional join points which the compiler marked as extended multiple times. However, in these cases conditional pointcuts were used due to the Statement Extension or Advise Objects Problem. These join points are never actually extended multiple times at runtime, but the compiler is not able to determine this statically.

One simplified example of a SJP where the order matters is given in Figure 6.1. The original code to remove a transaction contains extensions from two features: (1) the whole

```
1 public class TxnManager {
2   void unRegisterTxn(Txn txn) throws DatabaseException {
3     allTxnLatch.acquire();
4     allTxns.remove(txn);
5     getMemoryBudget().updateMiscMemoryUsage(txn.getInMemorySize());
6     allTxnLatch.release();
7   }
8 }
```

(a) Original code.

```
1  aspect privileged MemoryBudgetExt {
2    after(Txn txn, TxnManager mgr) : execution(void TxnManager.unRegisterTxn(Txn)) &&
         args(txn) && this(mgr) {
3      mgr.getMemoryBudget().updateMiscMemoryUsage(txn.getInMemorySize());
4    }
5  }
6  aspect privileged LatchExt {
7    around(TxnManager mgr) : execution(void TxnManager.unRegisterTxn(Txn)) && this(mgr) {
8      mgr.allTxnLatch.acquire();
9      proceed(mgr);
10     mgr.allTxnLatch.release();
11   }
12   declare precedence : LatchExt, MemoryBudgetExt;
13 }
```

(b) Refactored aspects.

Figure 6.1: Feature Order Example with SJP.

method is synchronized with latches in Lines 3 and 6, and (2) the memory budget is informed about each change in Line 5. We refactored this example with two ESMR refactorings into the two aspects *MemoryBudgetExt* and *LatchExt* as shown in Figure 6.1b. In this case the order in which the aspects are weaved decides whether the memory budget code is synchronized or not. Because we do not know if the synchronization is really necessary, we decided to enforce the stricter synchronized version of the original source code and added a *declare precedence* statement (Line 13)[1].

The only more complex example is the order in which the four daemon threads are created and shut down. As shown in Figure 6.2, the *Evictor* is created first, than the *Checkpointer*, than the *INCompressor*, and finally the *Cleaner*. However, the order in which these threads are shut down is different, almost reverse: first the *INCompressor*, than the *Cleaner*, than the *Checkpointer*, and finally the *Evictor*. To reimplement this order we declared the aspects precedence in their shutdown order and implemented all shutdown extensions with *after* advice. Thus, the shutdown method is extended in the correct order. To achieve the correct creation order we used other advice. The first two features *INCompressor* and *Cleaner* add the creation code with *after* advice while the last two features add their code

[1]Note: The mathematical concept of precedence has the opposite meaning of precedence in AspectJ. Higher precedence in AspectJ means apply later, whereas mathematical precedence means apply earlier (Lopez-Herrejon et al., 2006). Therefore, the precedence is defined in the order *LatchExt*, then *MemoryBudgetExt*.

```
1  public class EnvironmentImpl {
2    private void createDaemons() throws DatabaseException {
3      evictor = new Evictor(this, "Evictor");
4      checkpointer = new Checkpointer(this, cpWakeupTime, "Checkpointer");
5      inCompressor = new INCompressor(this, wakeupInterval, "INCompressor");
6      cleaner = new Cleaner(this, "Cleaner");
7    }
8    private void shutdownDaemons() throws InterruptedException {
9      shutdownINCompressor();
10     shutdownCleaner();
11     shutdownCheckpointer();
12     shutdownEvictor();
13   }
14   ...
15 }
```

Figure 6.2: Daemon Threads Creation and Shutdown Order.

with *before* advice. This strategy reproduces the correct creation order without the need for a different aspect precedence.

As seen in the previous examples, we used the *declare precedence* statement to specify the weaving order for aspects. However, we observed that it is sufficient to order features instead of aspects. In few cases only, a SJP is extended by two advice statements from the same feature, and in these cases the order does not matter. All other SJPs are extended by different features. Even in more complicated cases like the daemon creation and shutdown order shown above, we could express the necessary order with features. We did not need the ability to order individual aspects or even advice declarations.

Because we cannot directly declare feature precedence with AspectJ, we used the *declare precedence* construct as shown above to define an order for aspects where needed. This was convenient enough for our refactorings, however in larger projects these statements could easily be generated with a tool that transforms feature precedence into aspect precedence statements between all aspects in these features.

During implementation, we observed that the placing of these precedence statements is a problem. When this statement is placed in an optional feature it has an effect only when the feature is included. The placement of *declare precedence* statements is a special case of the Feature Optionality Problem (cf. Chapter 5). For a clean solution we would need to create derivative features for all possible dual feature combinations, i.e., the maximum number of derivative features as discussed above. Just to express the precedence for the 7 statistics features we would need 21 derivatives. Higher order derivatives are not needed because precedence declarations are transitive, thus dual feature combinations are sufficient.

For our refactorings, we used a trick to avoid derivatives and exploited the fact that it is possible to specify a *declare precedence* statement with non-existing aspects. The used AspectJ compiler generates only a warning, which we ignore. So, we created one *declare precedence* statement only for all statistic features inside the STATISTICS code, that is required by every statistics feature: *'declare precedence : INCompressorStats, EvictorStats, CheckpointerStats, CleanerStats, CachingStats, FSyncStats, LogStats;'*. Similarly, we placed one *declare precedence* statement for the daemon thread features in the base

code. However, this trick has two major disadvantages. First, because of scope limitations (cf. Section 4.2.6), it works only when all aspects are declared in the same package. Second, we added a *declare precedence* statement to the STATISTICS feature and the base code that otherwise would not need to know about the presence of other features. Thus, as also observed by Murphy et al. (2001), the declaration of precedence inside a feature breaks encapsulation.

For a clean and manageable solution, we suggest an approach similar to FOP: the feature order should be managed outside of the source code. A tool then creates all needed *declare precedence* statements based on the feature order before compilation. Thus, this tool would simplify this process and abandon the need to create a high number of derivatives.

To summarize, we observed some shared join points, although the percentage is very small. In all cases the AspectJ *declare precedence* statement could be used to specify the feature order. However, due to the Feature Optionality Problem the placing of these statements is a major problem.

6.2 Observations on Accidental Weaving

Several studies have shown potential problems of extensions with pattern expressions. These pattern expressions can accidentally extend join points introduced in subsequent development steps and thereby inflict unanticipated composition and unpredictable behavior in evolving aspect-oriented software (e.g., Lopez-Herrejon et al., 2006; McEachen and Alexander, 2005). Especially critical are *call* pointcuts that can extend call join points introduced by later development steps.

During our refactorings we did not observe any problem with accidental weaving, even though we sequentially refactored one feature after the other. We recognized several reasons for this:

- As described earlier we hardly used any homogeneous extensions that rely on pattern expressions. Those were simply not needed during our refactorings, so that the main potential cause of the accidental weaving problem is rarely present in our application.

- During early development phases we used pattern expressions instead of method signatures to simplify the writing of pointcuts (cf. Section 4.1.6). However, these pattern expressions usually specify the target class and the full method name and use patterns only for return type and parameters. This seemed to be sufficiently accurate to avoid accidental weaving because we rarely faced overloaded methods.

- We used *call* pointcuts mainly to emulate statement extensions (cf. Section 4.1.3 and 4.2.1). In these cases we usually restricted the pointcut with the *within* and *withincode* pointcut designators. This again reduced the chance for accidental weaving.

- Finally, we usually did not rename methods or changed there signature during refactoring. Thus, we did not really introduce new join points, but just moved existing join

```
1  public class Innocent {
2    public void someMethod() {
3      ...
4    }
5  }
6  public aspect Naughty {
7    before(Innocent a) : call(void Innocent.someMethod()) && target(a) {
8      a.someMethod();
9    }
10 }
```

Figure 6.3: Aspect Recursion (adapted from Forster and Steimann, 2006).

points in most cases. The accidental weaving problem was consequently avoided, because we were able to exclude unwanted join points when we created the pointcut in the first place. We assume that the accidental weaving problem is an evolution problem, not a refactoring problem.

This shows that, at least in our refactorings, accidental weaving is not an issue. However, during further evolution or maintenance we may still encounter these problems as described in the studies mentioned above. Nonetheless, the low amount of homogeneous pointcuts indicate that even than the accidental weaving problem will not occur frequently.

6.3 Observations on Aspect Extensions

Forster and Steimann (2006) discussed the problem that the unbounded 'at once' weaving can create logical problems and infinite loops when aspects extend code introduced by other aspects or itself. A simple example of such a problem is shown in Figure 6.3, where the *Naughty* aspect advises a call it introduced itself (Line 8). A similar problem can occur with cyclic extensions of two or more aspects, for example, if aspect *Foo* extends code introduced by aspect *Bar* and aspect *Bar* extends code introduced by aspect *Foo*. Forster and Steimann (2006) argues that an ordered bounded weaving prevents such problems.

In our refactorings we have not observed any of these problems. There are many cases where one aspect extends code introduced by other aspects as already shown in our dependency analysis in Section 3.5 (cf. Table 3.3, p. 51). However, in this analysis we have detected only one mutual dependency, where the MEMORYBUDGET feature extends the LATCHES feature and the LATCHES feature references the MEMORYBUDGET feature. Still, this mutual extension does not create any cyclic extensions and could easily resolved for an ordered weaving with bounded quantification, e.g., with the idea of sandwiching (Parnas, 1978; Apel and Liu, 2006).

In all other cases where aspect code is extended, the extension is unidirectional. Furthermore, no aspect extends join points created by itself. We could even generate an aspect order from the feature model and its dependencies. This observation shows that the possibility for logical problems and infinite loops due to the unbounded 'at once' weaving has no practical relevance, but on the other hand an bounded ordered weaving would not hinder

```
1 public class Base {
2   void main() {
3     ...
4   }
5 }
```

```
1 public aspect Foo {
2   void Base.lock() {
3     ...
4   }
5   void Base.unlock() {
6     ...
7   }
8 }
```

```
1 public aspect Bar {
2   void around(Base base) :
        execution(void Base.main()) &&
        this(base) {
3     base.lock();
4     proceed(base);
5     base.unlock();
6   }
7 }
```

Figure 6.4: Error Propagation.

refactorings, either.

6.4 Other Problems of AspectJ's Weaver

We observed two other problems that arise due to the 'at once' weaving approach: error propagation and restricted IDE support on projects with errors. The *Error Propagation Problem* occurs when the source code contains errors. Usually the compiler can output the position of the problem to help the developer fixing it. However, AspectJ weaves all aspects 'at once', thus an error can create many subsequent errors that make it hard to find the original one. The compiler cannot distinguish between original errors and subsequent errors which are caused only by other errors.

Consider the example of base code and two features in Figure 6.4. The aspect *Foo* introduces two methods into the base code while the second aspect *Bar* uses these methods to extend the base code. When there is a compilation error in the *Foo* aspect, e.g., a syntax error like a missing return type, not only this error is shown but also two subsequent error messages in *Bar* like *"The method lock() is undefined for the type Base, Bar.aj"*. Thus, a simple error in one aspect can cascade to a whole list of errors that are hard to trace and where the initial error is hard to find. During our refactorings we had situations where a single error resulted in over 100 reported errors!

In a system with an ordered bounded weaving process this error propagation is no longer possible. The compiler can stop after a feature that could not be compiled correctly. Even if the compiler continues it is easier to detect the cause, because the first error, in our case the syntax error in *Foo*, is usually responsible for the ones from subsequently weaved aspects.

Similarly, IDE support like the AJDT Eclipse plug-in requires full compilation without errors to be able to use any tools like showing pointcut matches. This means that during development, when we refactored a new feature we had to resolve all errors in this feature first, before we could use the IDE support again to see extensions by previous features.

Among other problems, this makes it hard to detect SJPs already during development. With an ordered weaving the IDE could still show all extensions made from previous features and thus ease development.

6.5 Summary

The ordering of features and aspects inflicted fewer problems than expected. Almost all features extend disjoint pieces of code, and potentially dangerous logical errors or infinite loops have no practical relevance. Due to the nature of our refactorings and the low number of homogeneous pointcuts, accidental weaving is not a problem either. In those few places where a join point is extended multiple times—only in 5 % of all extended join points—a simple rule of precedence for features is sufficient. Solely the definition of this rule with AspectJ constructs poses a problem.

Therefore, we conclude that the proposed extensions, like bounded aspect quantification or ordered weaving, are not necessary. On the other hand, these extensions are not hindering either, so their aim to gain a cleaner language and to avoid potential problems, even though not observed during our refactorings, should be sufficient to integrate them into the AspectJ language.

Chapter 7

Understandably and Maintainability of Refactored Features

During our refactorings we got the impression that the resulting features are unreadable and unmaintainable. Especially we observed the following five points:

- The AspectJ syntax is hard to read; it is hard to understand existing code and tool support is crucial.
- The AspectJ syntax is hard to write; it is a long and repetitive process to write a pointcut.
- The complexity of aspects implementing a feature grows quickly, making it hard to understand large features.
- A feature cannot be understood independently. It is coupled strongly with the base code and other features.
- Pointcuts are fragile. It is very hard to evolve and maintain a refactored application.

Even though our impression is hard to quantify and may be dismissed by some as a personal view, in this chapter we try to illustrate our perception.

7.1 AspectJ Syntax

First, we focus on the syntax of AspectJ. We feel that compared to other extension mechanism the syntax of AspectJ is very complicated. In this section we try to illustrate our impression, point out some specific problems of advanced pointcuts and scale, and propose metrics to measure complexity.

7.1.1 General Problems

Increased Code Size. An obvious observation is that AspectJ statements are usually very long in terms of code size. As illustration, consider this simple example: we want to extend the *delete* method in the *Database* class and introduce a trace call. In an object-oriented framework we would create a subclass of Database and override the delete method as shown in Figure 7.1a. However, in AspectJ we need at least three pointcut designators for the same behavior as shown in Figure 7.1b. First, we need an *execution* pointcut to capture the method's execution join point. For this pointcut designator we have to specify the method signature with the return type and all parameter types[1], unless we are willing to use pattern expressions for heterogeneous pointcuts, even though accidental weaving can occur (cf. Section 6.2). Second, we need an *args* pointcut designator to capture the needed arguments. Finally, we need a *within* pointcut designator to restrict this extension from affecting subclasses[2]. This extension is 94.5 % larger than the object-oriented implementation: the AspectJ version has 389 characters, while the object-oriented version has 200 characters, each not counting line breaks and unnecessary blanks. In our perception the AspectJ version is additionally much harder to read. Even when we use an anonymous pointcut as shown in Figure 7.1c, we still have an increase in code size of 39.5 %.

AspectJ advocates usually argue that the increased overhead by the AspectJ language constructs is outweighed by the code savings of homogeneous pointcuts that remove repeated code. Our experience has shown that this does not hold. As discussed earlier, only a small percentage of all extensions were homogeneous, and the homogeneous ones extend only few join points each with small code fragments. Furthermore, some of the homogeneous extensions use enumerations, thus resulting in especially large pointcuts. Exact figures are hard to collect, but we estimate that even the homogeneous extensions do not save much code size compared to the overhead they introduce. If the massive amount of heterogeneous extensions are taken into account as well, the balance is far negative: the overall code size grew by the usage of aspects.

Repetitive Parameter Lists. The AspectJ versions are not only larger, but also highly repetitive. While the object-oriented extension only declares the parameter list once and then uses it for a super call a second time, the AspectJ version with the explicit pointcut in Figure 7.1b repeats the parameter list 5 times: in the pointcut declaration (Line 2), in the execution and args pointcut designators (Lines 3–4), in the advice declaration (Line 6), and finally in the pointcut reference (Line 7). The AspectJ version with the anonymous pointcut in Figure 7.1c still repeats them 3 times. This repetition is laborious and error-prone and usually done with copy and paste.

[1] It is possible to specify thrown exceptions and also modifiers, e.g., *protected*, *static*, or *final*, in a pointcut. However, in Java it is not possible to overload methods with different exceptions or modifiers, thus we did not use them in pointcut declarations.

[2] Surprisingly, but described as intended behavior in the AspectJ manual, an *execution* pointcut affects the specified class and all subclasses unless it is restricted explicitly with a *within* pointcut explicitly. The pointcuts *'execution(void Database.put(..))'* and *'execution(void Database+.put(..))'* are equivalent.

```
1  public class TracedDatabase extends Database {
2    public void put(Transaction txn, DatabaseEntry key, DatabaseEntry data) {
3      super.put(txn, key, data);
4      Tracer.trace(Level.FINEST, "Database.put", this, txn, key, data);
5    }
6  }
```

(a) Object-oriented Extension

```
1  public aspect TracedDatabase {
2    pointcut tracePut(Database db, Transaction txn, DatabaseEntry key, DatabaseEntry data) :

3      execution(void Database.put(Transaction, DatabaseEntry, DatabaseEntry))
4      && args(txn, key, data) && within(Database) && this(db);
5
6    after(Database db, Transaction txn, DatabaseEntry key, DatabaseEntry data):
7      tracePut(db, txn, key, data) {
8      Tracer.trace(Level.FINEST, "Database.put", db, txn, key, data);
9    }
10 }
```

(b) AspectJ Extension with Explicit Pointcut

```
1  public aspect TracedDatabase {
2    after(Database db, Transaction txn, DatabaseEntry key, DatabaseEntry data):
3      execution(void Database.put(Transaction, DatabaseEntry, DatabaseEntry))
4      && args(txn, key, data) && within(Database) && this(db) {
5      Tracer.trace(Level.FINEST, "Database.put", db, txn, key, data);
6    }
7  }
```

(c) AspectJ Extension with Anonymous Pointcut

Figure 7.1: Syntax Comparison.

Programming in a Third-Person Perspective. What makes the AspectJ version additionally complicated is what we call the *third-person perspective*: The advice statement does not directly extend the target class, but is described as an external entity. It does not have direct access to the extended object and cannot use the *this* keyword to describe extensions like a first-person narrator would. Instead, extensions have to refer to the target class through an explicit object, captured with a pointcut.

In the object-oriented implementation of our example (Figure 7.1a) we simply use the *this* keyword to pass the extended database object to the trace method. Instead, the AspectJ variants have to add the database object to the pointcut's parameter list and intercept it with the *this* or *target* pointcut designators (Line 4 in Figures 7.1b and Figure 7.1c). Alternatively, the extended object can be accessed using the *thisJoinPoint* API, e.g., *'Tracer.trace(..., (Database) thisJoinPoint.getThis(), ...)'*, although because of the required dynamic typecast we did not use this version. It is necessary to emphasize that this access to the extended object is not a constructed biased example. In our refactorings 400 advice declarations—83 % of all—required a *this* or *target* pointcut designator to access the extended object. Examples can be seen throughout this thesis, e.g., in Figures 3.2–3.5, 4.1, 4.2, 4.7, 4.11, 6.1, and 7.1.

The necessity to capture the extended object with a pointcut makes not only the advice declaration larger, it also makes it harder to read the extension. In Java the *this* keyword is optional and it is common not to use it, unless it is required. It is common to expect that all calls or variables are members of the current class if no explicit target is specified. This is not possible in AspectJ. There the extended object must always be intercepted with a pointcut and specified explicitly.

We perceive this third-person programming perspective as unfamiliar, unusual, and—except for some homogeneous crosscuts—unnecessary. We much more often refer to the extended object than to the aspect's instance. The confusion is even increases because methods introduced with ITMDs are written in a first-person perspective, thus both perspectives are frequently mixed in one aspect.

7.1.2 Advanced Pointcuts are Hard to Understand

Even though the pointcuts shown in the above example are already fairly complex, their object-oriented equivalent is just a simple method refinement. Earlier in this thesis we have shown that AspectJ allows advanced crosscuts like conditional extensions, statement extension emulation, and homogeneous extensions. In this section we try to show why these extensions—in our perception—are even worse to understand.

Conditional Extensions. First, let us revise conditional extensions. In contrast to the simple method refinement above, conditional extensions are executed only depending on a runtime condition. This makes it harder to predict and understand their behavior. The developer must not only understand at which join points the extension applies, but also when the enabling condition can arise in the source code. Usually tools cannot support developers in this task.

We experienced that those extensions are often weaved into join points where they never occur. Recall the Advise Objects Problem (Section 4.2.5) where we used a runtime condition to extend only certain subclasses of the standard *Set* class (Figure 4.11e). Even though the actual extension occurs only in few join points where the subclass can ever be used, many other join points are also matched. When the developer tries to understand this extension he first has to visit all potential join points and then learn by reading the code, that only few of the potential join points will ever execute this extension. This might be obvious in some cases, but hidden as implementation detail in others. Here again, tool support is not possible, because a tool cannot statically decide which extensions are executed at runtime.

Control flow based extensions. As shown earlier, AspectJ allows special conditional extensions that depend on the dynamic control flow of the application. We have experimented with the *cflow* pointcut designator to work around some problems like the Statement Extension, Parameter Passing, or the Local Variables Access Problem. Already in Figures 4.4 and 4.9 we have shown two cflow solutions. We have argued in both cases that these solutions are hard to understand, but why is this?

A first reason that makes the cflow statement hard to understand is, that there is no equivalent known from object-oriented programming. Thus, developers do not only need to learn a new syntax but a completely new concept.

Another reason is—we assume—that a cflow statement increases the complexity from one pointcut to a combination of two pointcuts, and potentially even more, as arbitrary complex solutions with combined or nested cflow statements are possible. To understand what happens, the developer has to identify two independent sets of join points and try to understand their connection. This means he must not only find the right place in the source code, but must really comprehend the control flow of the base code. This might be fairly simple in some cases like the examples in this thesis, but when dealing with homogeneous pointcuts or control flows over several calls, we found it very hard to understand the constructs.

To be fair, there might be situations where the cflow solution is easier to understand than an object-oriented approach that solves the same problem. One is the frequently quoted observer example to avoid triggering an update inside another observed method: *'after(): displayStateChange() && !cflowbelow(displayStateChange()) { Display.refresh(); }'* (Kiczales and Mezini, 2005). However, we assume that these situations are delimitable problems with a small underlying code base. Furthermore, if they existed in Berkeley DB, they were already solved differently and thus did not occur during refactoring. More importantly, the potential to abuse the *cflow* pointcut as a workaround for other problems like passing parameters makes it dangerous to use at all.

Homogeneous Extensions. Also homogeneous extensions make it harder to understand an AspectJ program. Enumerated pointcuts are easiest to understand, they just make the pointcut definition larger. However, pattern expressions must either be very simple or require tool support to find all affected join points.

Additionally the developer needs to understand the intension of the pointcut. For example, we used the pointcut *'execution(void MemoryBudget.update*MemoryUsage(..))'* to

match the methods *updateTreeMemoryUsage*, *updateMiscMemoryUsage*, and *updateLockMemoryUsage*. However, the pointcut alone does not describe the intention, e.g., whether an new method *updateFormerMemoryUsage* should be extended or not. Instead the developer has to gasp the intention from the context: the operation of the advice declaration, the affected join points, or if available, the name of the pointcut. It is especially hard to tell if a homogeneous pointcut matches the correct join points, or too many or too few.

As explained before, the long syntax to specify a full method signature for *call* and *execution* pointcuts tempted us to use pattern expressions to simplify the writing of heterogeneous extensions at the beginning of our refactoring. Furthermore, we have observed this effect in other AspectJ projects as well (e.g., Apel, 2007). Pattern expressions for heterogeneous extensions reduce the overall code quality: A new developer trying to understand existing source code has to check for every single extension if it is homogeneous or if the pattern was just used for convenience. This might result also in mistakenly interpreting homogeneous extensions as simplified heterogeneous ones.

7.1.3 Complexity and Scale

Measuring Complexity. Previously, we have expressed our perception that pointcut definitions are complex. In this section, we try to support this view by providing metrics and measuring the complexity.

First, as a *basic metric*, we suggest to count the number of pointcut designators used for each advice after resolving named explicit pointcuts. So the lowest complexity is an advice statement that uses only one pointcut designator. The simple method extension example above in Figure 7.1 has a pointcut complexity of 4, because it uses the designators *execution*, *args*, *within* and *this*.

For further work, we suggest to use an *extended metric* that uses different weights for different pointcut designators and different combination of pointcut designators based on the empirical perception of their complexity. For example, we—again subjectively—perceive a pointcut with a *cflow* statement, or a *call* and *withincode* combination as much more complex than a simple *execution* pointcut. For the same reason, explicit pointcuts might be measured as more complex than anonymous ones. However, due to the lack empirical data we use the basic metric as approximation in this thesis.

In Figure 7.2 we show the measured pointcut complexity of Berkeley DB based on the basic metric. The simplest case with just one pointcut designator was only used 5 times, in all other cases multiple designators were connected with *&&* or || operators. Most frequently pointcuts with three or four designators were used. Especially frequent are combinations of *execution* or *call* with *args*, *this* and *within*/*withincode* designators, typically for method extensions. Pointcuts with more than 4 designators are less frequent and usually occur only for enumerated pointcuts. At average, 3.45 pointcut designators were used per advice statement. This again shows the high complexity of the necessary AspectJ constructs. Pointcut designators are rarely used isolated but usually in combination.

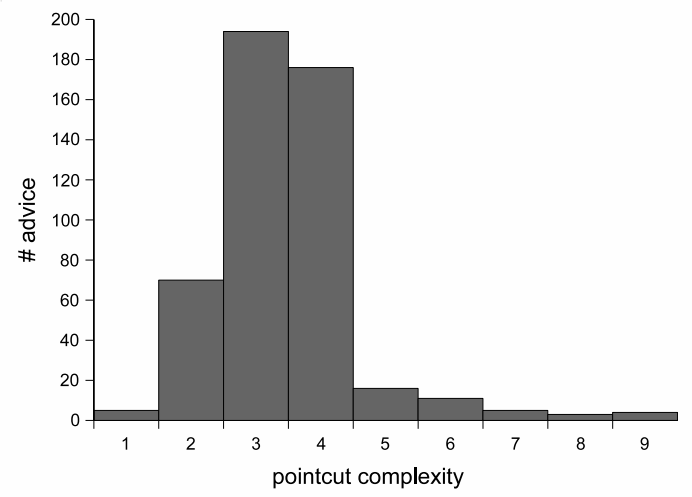


Figure 7.2: Pointcut Complexity Diagram.

Scaling Aspects. The examples used in common AspectJ books or papers are usually very small. Even the more complex examples like Transactions in Laddad (2003, p. 356ff) use only very few pointcuts and advice statements. In contrast, in Berkeley DB we refactored many medium sized and large features as shown earlier in our size analysis in Table 3.2 (p. 45). Many features have over 100 LOC and over 10 advice or ITMD statements. For example the LATCHES feature, one of the large features, has 104 pieces of advice and 51 ITMDs in 1835 LOC.

We observed that the small AspectJ examples and our small features are usually easy to understand. However, to read and understand a feature with 104 pieces of advice much harder. All refactored features implement only one design decision, e.g., synchronizing the implementation for multi-threaded operations, but the code size required to implement this design decision makes a huge difference in understandability.

Furthermore, we observed that in large features the advice statements are often similar. For example to implement the LATCHES feature, most of the 104 pieces of advice are *around* advice which first acquire a latch, than call *proceed* and finally release the latch. But even though many advice statements are similar, they vary in their concrete implementation depending on the context and the target class. Similarly, we observed that pointcuts in large features do not become significantly more complex, there are just more pointcuts (cf. Figure 7.3). Thus again, refactoring large features like LATCHES is a highly repetitive tasks.

Whether this complexity is inherent feature complexity or caused by the language used for the implementation is disputable and must be deferred to further work. Our perception is that AspectJ as language make it harder to understand features.

It seems to us, that the drawback of the hard to read AspectJ language can be justified only when a small aspect is used homogeneously. The larger aspects become, the harder they are to understand and maintain. To understand a small aspect it is possible to read the whole source code and try to figure out how the advice statements work together, step by step. Our small features are also fairly easy to understand. However, to do the same in a

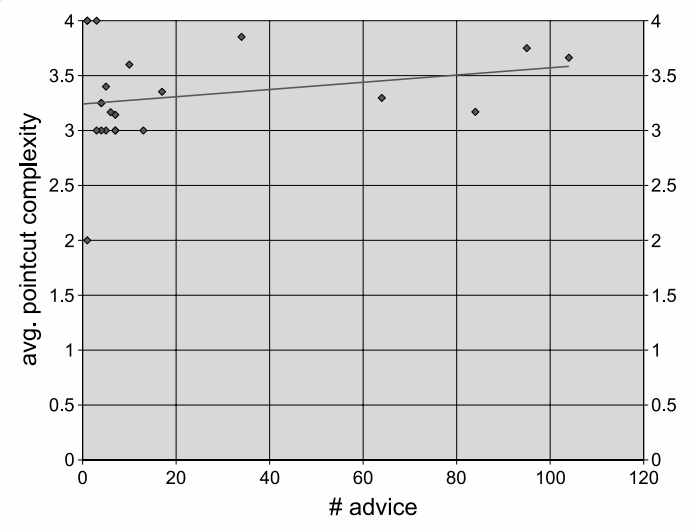


Figure 7.3: Feature Size vs. Pointcut Complexity.

large aspect is challenging.

Experience in Related Work. The complexity of advice and pointcuts is a well known problem. Some researchers avoid using AspectJ constructs wherever possible and just use aspects only to 'glue' objects together. For example, Murphy et al. (2001) use classes instead of ITMDs and manipulate the inheritance hierarchy or advise only few join points to call these classes:

> *"Since in the AspectJ case, we separated most of the code into classes rather than aspects, the end result may also be more straightforward to understand. Maintenance may be easier [...] because the aspect, which contains the code connecting the base and separated structures, is short and straightforward."* (Murphy et al., 2001, p. 280f)

Thus, they use AspectJ to emulate mixin-based inheritance (cf. Bracha and Cook, 1990) to avoid the complexity of pointcut definitions. Similarly, Monteiro and Fernandes (2005) argue for the use of normal methods and member variables inside aspects instead of ITMDs to introduce an equivalent behavior.

7.2 Coupling of Base Code and Aspects

We observed a strong coupling of the aspects with base code and other aspects. We assume this is because of a fine granularity of our extensions.

Most common AspectJ examples implement non-critical concerns like tracing, caching or policy enforcement and they often use very general pattern based pointcuts like "advise all non-static methods with tracing code". Furthermore, many authors, e.g., Laddad

(2003), Sullivan et al. (2005) or Filman and Friedman (2005), suggest to extend only public interface methods where possible, because these methods are stable to changes and the implementation of the class remains hidden.

Our aim was to refactor design decisions into optional features. When we identified features, we did not specifically search for features that fulfill the above criteria. During refactoring we did not observe any of these highly repetitive homogeneous crosscuts that depend on existing interfaces only. Quite contrary, the design decisions we refactored were typically encapsulated in classes or existing collaborations of classes. Consequently, our extensions use a very fine granularity: we frequently extended internal methods or even statements and heavily depend on implementation details.

This reliance on implementation details creates a tight coupling between the base code and the aspect (or between two aspects if one aspect extends join points of the other). A special problem with AspectJ is that this coupling is implicit and invisible for the developer. In contrast to a class refinement, which has a limited scope and can only affect protected or public methods in super-classes, an aspect can extend *any* join point in the application, even in private and internal methods. The base code contains no hint that it is extended by aspects—it is oblivious to the aspects—and changing it without adapting the aspects may break the application. *One can no longer change the base code or the aspect without risking to break the code*. All changes to the base code, even local changes in private methods, require an understanding of all aspects (currently 151 in Berkeley DB) that may extend the code.

To evolve and maintain source code developers therefore usually depend on tool support, like an IDE that highlights extended join points. Such tool support is already used in Java development to navigate complex class hierarchies, however with AspectJ tool support becomes crucial to understand all the source code. The AJDT Eclipse plug-in is such a tool. However, its usage is limited to code without errors, what is not always given during development (cf. Section 6.4). Additionally, it requires explicit reweaving frequently during source code editing to indicate extensions to newly added join points. Finally, the AJDT plug-in is unable to correctly show extensions to join points introduced by other aspects, which we frequently used. It even shows errors when calling a method introduced with an ITMD by an aspect, even though the code is correct and compilable. As development with AspectJ heavily depends on tool support, such problems are not acceptable. Requirements must be higher for AspectJ IDEs than for Java IDEs, because in the latter the developer can still manually navigate and overview the effect of his changes himself.

We conjecture that mechanisms that ensure an encapsulation and allow aspect extensions only through certain interfaces of the base code, like Open Modules (Aldrich, 2004) or Design Rules (Sullivan et al., 2005), could improve the maintainability in our case. They do not reduce the coupling, but at least make it explicit and thus easier to understand. Nonetheless, such approach would force us to publish the former encapsulated and internal code as interfaces. The advantage of the original object-oriented encapsulation, namely to be able to change implementation detail, is given up for external extensions by aspects. This also means that the base code has to be prepared for the extensions, again giving up obliviousness.

To summarize, the tight implicit coupling between the base code and aspects is a serious problem. Neither base code nor aspects can change without an understanding of the full source code. The main approach to solve this problem seems to be tool support in AspectJ IDEs. However current tools are not sufficient for the high demands and cannot guarantee a safe evolution.

7.3 Fragile Pointcuts

In the previous section we have shown the tight coupling of aspects with the base code. Aside from making the source code harder to understand, it also makes the source code very fragile. In this section we focus on the well known *Fragile Pointcut Problem* (Koppen and Störzer, 2004; Stoerzer and Graf, 2005) which we frequently observed during our refactorings.

7.3.1 Problem Definition

The Fragile Pointcut Problem is a software evolution problem, that occurs when an aspect-oriented application is changed, but the pointcuts are not updated accordingly. If this happens, the behavior of the application can silently change, because aspects advise different join points than before. In some cases the change is obvious, in others it may be subtle and go undetected. Stoerzer and Graf (2005) have identified four source code changes that are common during maintenance and evolution which modify join points and are thus potentially dangerous: (1) renaming a class or method, (2) moving a class or method, (3) adding or removing a class or method, and (4) changing a method's signature, e.g., by introducing a new parameter. Any of these changes modify the system's join points and may require updates to aspects.

The problem lies in the fact, that the developer often is not aware of aspects. As mentioned above, he either needs a tool that warns him or must understand all aspects in the system. Contrasting changes in object-oriented systems, where the compiler issues a semantic error when only the methods definition has changed but not the calls, in AOP those changes can go undetected, the compiler silently does not weave the join point.

The Pointcut Fragility Problem is especially dangerous for homogeneous pointcuts and pattern expressions[3]. We expect heterogeneous extensions to extend exactly one join point, it is therefore easy to check correct weaving. The AspectJ compiler even issues a warning when advice is not woven anywhere. In contrast, homogeneous extensions are woven at multiple join points, where the expected number is never specified. The compiler can therefore not issue a warning on changes. Pattern expressions like *'execution(* Database.put*(..))'* might also create an illusion of stability against changes, however this makes it even harder to detect exceptions and fosters accidental weaving (cf. Section 6.2).

[3]Concerning pattern expressions the Fragile Pointcut Problem is also known as *Arranged Patterns Problem* (Gybels and Brichau, 2003).

During our refactorings we observed frequently that pointcuts written for previous features did no longer apply anywhere. We often only found the problem because of the compiler warning when we edited the aspect in a later development step or when we reviewed it for this thesis. In these cases we had to search for the cause of the change and fix the pointcut. Some changes in the few homogeneous pointcuts might even have happened undetected, although we tried to check all homogeneous pointcuts once more in a final step. The most frequent causes for the Fragile Pointcut Problem we observed in Berkeley DB were (1) signature changes, e.g., when we added another parameter to a hook method to grant access to a local variable, (2) object-oriented refactorings we made to create new join points, e.g., for the EIMR refactoring, and (3) moving code from a method to advice and thus either moving or removing join points.

7.3.2 Solutions

There are various proposals to avoid the Pointcut Fragility Problem. They tackle the problem with different approaches which reach from external test mechanisms, over aspect-aware refactorings, to new pointcut languages. In the following we give a very brief overview and evaluate the solutions on the background of our experience.

First, it is possible to use *test suites* to ensure a correct weaving as discussed earlier in Section 3.3.5. Unfortunately, this implies that there must be a unit test for every pointcut and creating and maintaining such test suite requires a high effort.

A common solution is to *raise the developer's awareness* of extended join points. This can either be done by tool support or by placing pointcuts locally. For example, an aspect-aware IDE can mark code that is extended by aspects. The Eclipse plug-in AJDT (Clement et al., 2003) is such an IDE for AspectJ. However, existing IDEs we know of are not robust enough for practical usage. For example, AJDT shows only pointcuts in fully compilable code, and it always needs to rebuild the application to update any markers. During our refactorings this has turned out to be unreliable and slow. One approach that places pointcuts locally is *Open Modules* (Aldrich, 2004, 2005). In Open Modules pointcuts are located in the module they extend. Both approaches just raise the developer's awareness, they treat the symptoms but they do not solve the problem. The matched join point set of pointcuts can still change silently.

Another tool driven approach to avoid the Pointcut Fragility Problem is to use automatic *aspect-aware refactorings* for all changes to the source code (Hanenberg et al., 2003). Unfortunately, there is no such refactoring tool that can be used productively yet. Furthermore, many aspect-oriented refactorings cannot be automated because they cannot gather the intension of aspects, e.g., whether they should match new join points. Such a tool could also not decide automatically, just based on the static structure of the program, whether a conditional extensions like a *cflow* pointcut must be updated after a change.

Finally, the main objection against using aspect-aware refactorings is, that changes a refactoring tool must apply to a pointcut can be quite complex and can render a pointcut completely unreadable. Consider the example in Figure 7.4 where a simple Extract Method refactoring renders a simple pointcut completely unreadable. The mechanics of this refac-

```
1 class C {
2   void foo(int a, int b){
3     goo();
4     System.out.println(a + b);
5     this.x = a;
6   }
7 }
```

```
8 pointcut x():
9   withincode(void *.foo(..));
```

(a) Base Code and Aspect before Refactoring.

```
1  class C {
2    void foo(int a, int b){
3      goo();
4      bar(a, b);
5    }
6    void bar(int a, int b){
7      System.out.println(a+b);
8      this.x = a;
9    }
10 }
```

```
11 pointcut x():
12   (withincode(void *.foo(..)) &&
13     !call(void C.bar(int, int))) ||
14   (withincode(void C.bar(int, int)) &&
15     !execution(void C.bar(int, int)) &&
16     cflow(withincode(void C.foo(..))));
```

(b) Base Code and Aspect after Automated Extract Method Refactoring.

Figure 7.4: Aspect-aware Extract Method Refactoring (Hanenberg et al., 2003).

toring are described in detail by Hanenberg et al. (2003). In a nutshell, to be behavior-preserving, the refactoring must exclude the two newly created join points, i.e., the call and the execution join point for the *bar* method, from the pointcut, and it must prevent to match external calls to the *bar* method by restricting the pointcut with a *cflow* pointcut designator. This example illustrates that, even though automated refactorings might be possible, the refactorings make the resulting code more complex. A pointcut definition in the already hard to understand AspectJ pointcut language can become completely unreadable.

All in all, we estimate that aspect-aware refactorings can be used in some cases. Especially for the creation of hook methods and Extract Method refactorings, such refactoring tool could have been a good help during our refactorings. However, it is not a general solution for the Fragile Pointcut Problem.

Another group of tools has been suggested to compare the pointcuts between two development steps (Koppen and Störzer, 2004; Stoerzer and Graf, 2005; Clement et al., 2003). Such *pointcut delta analysis* can show which pointcuts match a different set of join points than in the previous steps and thus might show pointcuts that were not updated correctly. Such tools can drastically reduce the danger of fragile pointcuts when applied accurately and attentively. However, it has some practical disadvantages, because it reports many 'false positives' and requires the attention and diciplin of the developer to check the deltas frequently and carefully. It is probably more helpful during normal program maintenance than during refactoring, when it does not report so many 'false positives'.

All approaches so far only treat the symptoms of the Fragile Pointcut Problems, they do not tackle the problem: the fragile pointcut language. The creation of a less fragile pointcut language that avoids the problems is an important research topic today (Stoerzer and Graf, 2005). There are several suggestions for new pointcut languages. For example, Masuhara and Kawauchi (2003) introduce a data flow pointcut to AspectJ, and various researchers

(e.g., Gybels and Brichau, 2003; Rho and Kniesel, 2004; Ostermann et al., 2005) suggest to use logical programming and to reason over the static or runtime structure of the application to match join points. All these approaches have in common that they introduce new statements or a new language to reduce the necessity to lexically match names or source locations. These approaches differ a lot from the AspectJ pointcut language used in our refactorings. Evaluating these language on refactoring Berkeley DB would require to redo the refactoring. Therefore, we defer the evaluation of new pointcut languages to further work.

7.3.3 Summary

To summarize, the Pointcut Fragility Problem is a serious maintenance and evolution problem, we observed during our refactorings. There are various potential solutions, reaching from disciplined programming, over tool support, to new programming languages. However, none of the approaches is satisfying for our work on Berkeley DB. This shows that aspect-oriented applications with AspectJ in its current version are potentially dangerous to maintain. Special caution has to be taken to not break existing behavior while evolving the application.

7.4 Readability of the Base Code

A final question is whether the quality of the base code improves. Due to the feature-oriented refactoring the base code should get smaller and contain only general code that does not belong to any feature. This should improve the readability and maintainability of the base code.

In fact, the base code size got smaller and in some parts—for example where logging and synchronization code has been removed—easier to read. However, during our refactorings we frequently introduced hook methods or used the Extract Method refactoring to create artificial join points. Additionally, unused parameters are left in the base code or empty parameter context objects were introduced. The resulting base code therefore partly looks ‘strange’, it contains many constructs a developer would not use otherwise. Hence, we cannot claim an obvious result whether the readability of base code has improved, but must defer this to an empirical study in further work.

Again, it has to be kept in mind that the base code can no longer be changed without understanding all existing aspects, or an aspect-aware refactoring tool. So, the readability of the base code might have been improved, the maintainability and evolvablility have not.

7.5 Summary

In this chapter we have shown why we experienced the refactored features as hard to understand and hard to maintain. The AspectJ language is very complex and requires long, repetitive statements that are hard to read. Especially advanced pointcuts and large features

are hard to understand. Additionally, the source code is coupled strongly with the aspects and makes it hard to evolve them independently. Furthermore, because this coupling is implicit and hidden, the AspectJ pointcut language is very fragile, so that evolution and maintenance changes can accidentally change the applications behavior undetectedly.

In this chapter, we could show only our subjective perception, however in further work we plan to empirically analyze the understandability and maintainability of AspectJ, other AOP languages and other approaches to implement features.

Chapter 8

Related Work

The case study presented in this thesis relates to work from several fields of software engineering. In this chapter, we give a brief overview over related languages, studies, and concepts.

Implementing Features. There are various proposals and languages how to implement features. Using an aspect-oriented language as in this thesis is only one possible implementation of features.

Batory et al. (2004) advocate the use of *mixins* (Bracha and Cook, 1990) to implement features. Mixin-based inheritance is implemented for the Java language in the AHEAD Tool Suite as the *Jak* language. Similarly, *FeatureC++* also uses mixins to implement features (Apel et al., 2005).

Furthermore, Batory et al. (2004) and Apel (2007, pp. 61) list many more concepts to implement features, for example *layers* (Batory and O'Malley, 1992), *feature modules* (Kang et al., 1990; Prehofer, 1997), *subjects* (Harrison and Ossher, 1993), *virtual classes* (Madsen and Moller-Pedersen, 1989; Ernst et al., 2006), *nested inheritance* (Nystrom et al., 2004), and *classboxes* (Bergel et al., 2005).

Moreover, there are some approaches that integrate aspects with a different approach like collaborations to implement features. This aims at still being able to use aspects for homogeneous crosscutting concerns but to use other language constructs to define other extensions. Examples are *aspectual mixin layers* or *aspectual feature modules* (Apel et al., 2006c; Apel, 2007), *aspectual components* and *aspectual collaborations* (Lieberherr et al., 1999, 2003), *caesar* (Aracic et al., 2006), and *object teams* (Herrmann, 2002).

Finally, it is possible to use object-oriented frameworks or component infrastructures. Frameworks usually provide explicit extension points that can be used by features. They also frequently use design patterns (Gamma et al., 1995) like the *Observer Pattern*, the *Adapter Pattern*, or the *Strategy Pattern* for an extensible and configurable interface. A popular example is the Eclipse IDE, that is highly extensible and supports a flexible configuration. Different IDEs with different features can be composed by different plug-in selections.

Features with AOP. There are various related publications that deal with features implemented in an aspect-oriented language, although they are often named differently.

Griss (2000) first suggested to use aspects to implement features, even though there was no practically usable aspect-oriented language back then. An early study on decomposing features with aspect-oriented languages including AspectJ was performed by Murphy et al. (2001). This study focuses on expressibility of language constructs and has already found some of the limitations we observed in our refactorings, i.e., the need for hook methods and the hard to read aspects. Similarly, Lopez-Herrejon et al. (2005) noticed problems in using aspects to implement features because of a lack of infrastructure and means to describe coherent features containing multiple aspects.

Colyer et al. (2004) discussed how features could be separated with AspectJ to create program families. The discussion is held on a theoretical level, how FOP can help in separating concerns. They defined orthogonal aspects, i.e., features that can be composed independently and do not have any interactions. However, their definition of orthogonal features is very strict and avoids the Feature Optionality Problem and therefore seems hardly usable for practical refactoring.

A brief discussion about the quality of the resulting features with AspectJ was already initiated by Nyssen et al. (2005), based on the obliviousness and the implicit extensions of aspects. Based on a very small case study, the authors found that novel aspect-oriented mechanisms are not required for feature implementation and suggest tool-driven composition of object-oriented feature modules as better alternative.

Case Studies. There are also some empirical case studies on refactoring existing applications, mainly from the areas of embedded database engines and middleware systems.

1. The work closest to ours is the aspect-oriented refactoring of the embedded database engine *HSQLDB*, done as case study by Eibauer (2006). In this case study nine features are refactored into aspects. The main difference is that the feature selection was based on a catalog of typical crosscutting concerns that are supposed to be encapsulateable with AspectJ easily. Even though it focused especially on crosscutting concerns the results are similar disappointing: hardly any homogeneous crosscuts were used, most the advanced language constructs were not needed. Furthermore they observed a reduced readability, a strong coupling, and some similar language limitations like the Local Variable Access Problem.

2. Tesanovic et al. (2004) have refactored the C version of Berkeley DB into four features with an aspect-oriented language. This study shows the general possibility and the advantages of having a configurable version of Berkeley DB from a database perspective. They do not describe their refactorings further and do not focus on the feature implementations and their quality.

3. Another refactoring of an embedded database engine into features implemented with AspectJ was performed by Godil and Jacobsen (2005). The target was the very small embedded in-memory database engine *Prevayler*. They created six small features

and observed an improvement in code size and readability of the base code, but they actually increased overall code size and used almost only heterogeneous extensions. They analyzed the quality of the resulting features by source code size and explicit coupling.

4. Zhang and Jacobsen (2003, 2004) discuss the refactoring of five selected features of a large middleware platform into aspects. They call this process *horizontal decomposition*, and their aim is to configure the middleware platform by including or excluding features. However, they focus only on some selected crosscutting concerns found with Aspect Mining tools.

5. Similarly, Hunleth and Cytron (2002) extracted 21 features from a small middleware platform to make it configurable. They use an own infrastructure with distinct feature directories build on *ant*, which is similar to our solution (Hunleth et al., 2001). They focus on the differences in footprint and performance compared to optional feature implemented with object-oriented design patterns and observe improvements.

6. Coady and Kiczales (2003) refactored four small crosscutting concerns in *FreeBSD* into features, three of which are optional and can be used to configure *FreeBSD*.

All these case studies are smaller than our refactorings of Berkeley DB: Even though some case studies (e.g., 1, 2, 6) have a large code base in terms of LOC, only few features (usually 4–9) were refactored. These features affect only a small part of the applications, and feature models are very simple. No case study analyzed the effects of scale. Furthermore, all case studies except 2 and 3 focused on refactoring (homogeneous) crosscutting concerns. They only refactored features that are expected to perform well with AOP and made no comment about how to implement other features. The quality of the resulting feature implementations is usually evaluated only based on code size or explicit coupling. Readability, fragility, or implicit coupling are not considered.

The evolution of features implemented with aspects was studied only by case study 6. The study focused on software evolution by comparing three major releases of *FreeBSD* and has observed an improvement of the evolution with aspects instead of the original implementation. Compared to the whole source base, the features are very specific and small, i.e., only affected few files and have only one or two pointcuts each. Furthermore, they do not interact. The Pointcut Fragility Problem or Accidental Weaving Problem were not likely to occur, because all 4 small aspects have been carefully adapted in only two large explicit steps instead of a constant evolution by oblivious developers.

Aspect Mining. Aspect Mining is the field of research that focuses on finding crosscutting concerns automatically, especially scattered code in existing applications (Hannemann and Kiczales, 2001; Loughran and Rashid, 2002; Griswold et al., 2001; Bruntink et al., 2005). It was used in some related case studies to identify features (Eibauer, 2006; Zhang and Jacobsen, 2003). However, Aspect Mining is usually performed with the aim to improve the quality of an application by replacing scattered code with aspects. This is different from

the refactoring into features, because scattered code must not represent a feature and, as we have seen, most features do not use homogeneous crosscuts and thus do not create scattered, but only tangled code. For an overview of Aspect Mining approaches see Bruntink et al. (2005), for an evaluation on finding features in an embedded database see Eibauer (2006).

Constructor Problem in C++. The Parameter Introduction and Passing Problem analyzed in Section 4.2.2 is similar to the to the Constructor Problem of mixin-based programming in C++ as described by Smaragdakis and Batory (2001) and Eisenecker et al. (2000). They describe the problem that some mixins introduce new parameters to the constructor of a class. The Parameter Introduction and Passing Problem is more general and includes the Constructor Problem. Eisenecker et al. (2000) suggest a solution that requires complex C++ templates not available in Java or AspectJ.

Criticism of Object-Oriented Programming. When object-oriented programming pervaded software engineering about twenty years ago, there were publications that claimed that object-oriented programming is harder to read and to understand than traditional procedural approaches. For example, Carter (1994) reflects about the design of the *ADA* language and argues that subclasses are harder to read and to understand than procedural implementations that do the same, because the developer needs to look up the superclass to understand it. Object-oriented programming thus breaks locality. Another example is Snyder (1986), who argued that a weak encapsulation in some language implementations of inheritance could lead to a decreased understandability.

This sounds familiar to our criticism of AOP in Section 7.1. The suggested solution by Carter (1994) is to use subclassing only when the superclass can be found easily, similar to the Open Modules approach by Aldrich (2004), which suggests that the pointcuts should be located locally inside the target module. Furthermore, tools and IDEs are used in both cases to overcome these problems. Most modern IDEs support easy navigation between classes in a hierarchy. Similarly, AJDT allows navigations between advice declarations and advised join points.

History has shown that most developers do not mind the acclaimed ‘reduced readability’ for the improvements gained by object-oriented languages. Nevertheless, we still think that there is a big difference between having to look up a superclass and searching the whole source code for affected join points, but such old criticisms might provide an alternative perspective. This can be especially useful for further work on measuring complexity. Moreover, another difference to the introduction of object-oriented programming is that we have various proposals how features can be implemented, it is not a choice ‘AOP or no features at all’.

Furthermore, it took a while to develop guidelines on how to write procedural programs (e.g., Parnas, 1972). The same process took place for object-oriented systems, e.g., the *Law of Demeter* (Lieberherr et al., 1988). Similar work is currently discussed in the AOP community (e.g., Kiczales and Mezini, 2005) and might also resolve some of the readability and understandability issues in the long run.

Chapter 9

Conclusion

In this thesis, we have presented a case study for refactoring a legacy application into features with AspectJ. We created a feature model for the embedded database engine Berkeley DB and refactored 38 features. We are able to compose different sets of features to create over 12 million different tailored versions[1]. Although we were able to implement and compose features with AspectJ, our observations suggest that AspectJ is not an appropriate language for this task.

A closer look at the used language constructs revealed that only a small portion was used, mainly static introductions and method extensions. Advanced language constructs like conditional extensions or homogeneous extensions were hardly needed. On the other hand, we found several serious limitations that hindered a straightforward implementation of feature models, so that our refactorings had to use various workarounds like hook methods or hacks based on complex pointcut expressions.

A major problem of using AspectJ to implement features is the readability and maintainability of the resulting code. We found features implemented as aspects hard to read and hard to understand, especially as the size of the feature grows. It seems that feature implementations in AspectJ scale poorly. We could not even reveal a clear improvement of the base code which the aspects were detached from. Additionally, we have shown that the resulting code is extremely fragile and a verification of correctness is very hard. Approaches to avoid or control the fragility require usually an immense overhead or tool support in a degree that is not available or practical, yet. Finally, a strong implicit coupling between base code and aspects and between aspects themselves also reduce apparently maintainability and require strong tool support, that is not available sufficiently, yet.

Furthermore, AspectJ does not provide any of its own infrastructure to implement and manage features. We had to impose the AHEAD infrastructure. Known problems like the Feature Optionality Problem occur also when features are implemented in AspectJ, but the language does not provide new solutions, even though one approach seemed promising at first. Various refactorings have been published, usually as informal descriptions, there is no tool that could be used for (semi-)automated refactoring in our project so that we had to do

[1] After removing all implementation dependencies with derivative modules, it is possible to create over 16 billion different tailored versions.

all refactorings manually anyway.

The ordered weaving of aspects repeatedly suggested in recent research can be applied in our case study surprisingly easily. We did not observe any of the problems ordered weaving tackles though. Thus the proposed extensions, like bounded aspect quantification or ordered weaving, were neither necessary nor hindering in our case study. Consequentially, this supports these extensions in creating a cleaner language which avoids potential problems in the first place.

Overall, AspectJ does not seem to have any unique advantage except the rarely used mechanisms to define homogeneous or conditional extensions. At the same time, it creates source code that is hard to read, fragile, and hard to maintain or evolve. We strongly suggest to evaluate other approaches to implement features.

Perspective. At first, our results seem contradictory to the ones of previous studies which reported positively about refactoring legacy applications into features with AOP. To align our work with them, we need to distinguish two separate topics, often mixed in these discussions and evaluations: modularizing (homogeneous) crosscuts and implementing (optional) features.

The positive results in prior work usually focus on improvements of the encapsulation of, usually small and homogeneous, crosscuts, i.e., encoding scattered and tangled code in one aspect. That some of these crosscuts can be used as optional features to configure an application is a byproduct, not a goal. However, when focusing on implementing (large-scale) features to create a configurable application in general, it is frequently observed that most features make heterogeneous extensions. While it is still possible to implement such features with AspectJ as shown in this thesis, the question is whether carrying forward the positive results from modularizing crosscuts to a general feature-oriented refactoring is possible. Discussions about implementing features must focus on expressiveness, understandability, and maintainability or evolution.

In this thesis, we could not evaluate if AspectJ is suited to encapsulate homogeneous crosscuts, because we rarely found any (probably because we did not specifically search for them with Aspect Mining tools). However our results for refactoring Berkeley DB show problems in regard to expressiveness, understandability, and maintainability which let us doubt its suitability to implement features in general.

This strongly emphasizes that the discussion about aspect-oriented refactoring of legacy applications must be split into separate discussions about modularizing (homogeneous) crosscuts and implementing optional features. If aspects are really required to modularize crosscuts as frequently claimed, than an integration of aspects for crosscuts and collaborations or other approaches to implement features seems favorable. Several recent publications propose such integrations (e.g., Apel et al., 2006c; Apel, 2007; Aracic et al., 2006).

9.1 Further Work

In this thesis we have opened and suggested various research topics for further work. First, there are various smaller open points in this thesis that could be examined in further work.

- The exception handling in AspectJ was not analyzed further in this thesis because the sample size was too small. We encourage further research to evaluate the possibilities to handle exceptions in aspect-oriented applications.
- The Parameter Introduction and Passing Problem we observed is a general problem of feature-oriented refactoring and not solved in any language we are aware of. Further research is required to evaluate the different proposals and—if necessary—to design appropriate language support to solve this problem.
- The analysis of the Feature Optionality Problem has shown that there is no satisfying solution, yet. The current approaches are either very complex, especially on large feature models, or they are inconvenient to use for the developer or end-user. Further work might include the evaluation of different languages, the design of a language for the Optional Weaving Approach, or means to avoid and hide complexity in the Derivative Feature Approach.
- In this thesis, we did not aim at measuring code footprint or performance. Further work might concentrate on such measurements and evaluate the advantage or penalty of AspectJ over the original object-oriented version. Similarly, the advantage of being able to build customized versions of Berkeley DB can be measured to give further insights whether a configurable database system is worth the effort required for refactoring and maintaining separate features.

Furthermore, we have found some larger topics that may provide valuable insights in further work:

Comparison of Pointcut Languages. In this thesis we have shown that the pointcut language of AspectJ is fragile, and that—even though various tools exist to cope with this fragility—a new pointcut language is necessary to not only treat symptoms but to solve the Fragile Pointcut Problem. There are various suggestions for new or extended pointcut languages (e.g., Masuhara and Kawauchi, 2003; Gybels and Brichau, 2003; Rho and Kniesel, 2004; Ostermann et al., 2005) that do not rely on lexical comparisons. Further research is necessary to evaluate their usability and fragility for practical refactoring and evolution.

We recommend to refactor some features of Berkeley DB once more with each proposed language to compare the results. Such a study can give further insights toward a solution for the Fragile Pointcut Problem and can help in selecting the best language for a given problem.

Comparison of Feature Implementation Approaches. As shown above, there are various proposals and languages to implement features. In this thesis we have evaluated only aspects as one approach, with AspectJ as one possible language. The results show that AspectJ is not a good choice, because its advanced capabilities are hardly used and because of various problems, many of which are specific to the AspectJ language. The question whether other AOP languages or different approaches like collaborations are better suited must be analyzed in further work (cf. Murphy et al., 2001; Lopez-Herrejon et al., 2005; Mezini and Ostermann, 2004).

The most important language constructs we have found were static introductions, method refinements and statement extensions. Static introductions and method refinements are elementary constructs supported by all of the approaches listed as related work in Chapter 8, while statement extensions are not yet common. A broader comparison can give further insights on which language constructs are required in a language and which is the best way to express them.

Empirical Study on the Cognitive Distance of the AspectJ Syntax. In Section 7.1 we illustrated why we felt that the AspectJ syntax is hard to read and understand. We could not underline it with an empirical proof. We therefore suggest to analyze empirically the cognitive distance of AspectJ to find causes and suggest improvements. Furthermore, such an empirical study can also give valuable insight when compared with to other approaches for the implementation of features.

Similarly, such a study should also consider the readability of the resulting base code. The AspectJ solution introduced some hook methods and other constructs which gave a 'strange' look to the base code. Other languages might provide a solution or have the same problem. An empirical study can analyze the benefits of the refactorings in terms of readability and maintainability of both, features and base code.

Finally, as proposed above, a comparison to the perception of the readability and complexity of object-oriented programming when it was proposed twenty years ago may help to adjust perspective and metrics.

Scaling of Feature Models. We analyzed the effects of scale on feature models in this thesis. We have shown that the current approaches dramatically increase complexity. The number of derivatives rises fast, quadratically to the number of features. This shows that research is needed on how to deal with feature models with a large number of features, e.g., to find suitable tools, languages, or infrastructures.

Furthermore, support is required for the end-user who composes the application. Our refactorings created 38 features (not counting derivatives). Already for these 38 features it can be hard to decide which features to include for a specific use case. Hence, important questions for further research are: How can a user select from 100 or 1000 features with thousands of dependencies? What kind of tools are needed to support the user?

9.2 ColoredIDE

During the refactoring of Berkeley DB we frequently reached the limits of AspectJ as described in this thesis. Discussions about other technologies to decompose Berkeley DB into features showed that also those would have similar problems. Therefore, we searched for a completely new approach to refactor a legacy application and implemented a prototype of a tool called *ColoredIDE*, which we intend to pursue in further work.

This tool follows the approach used in this thesis and other publications to illustrate how features are located legacy code with different colors. For example, in Figure 9.1 (cf. Figure 6.1a) we use two colors to show how two features affect the original *unRegisterTxn* method. Similarly, in ColoredIDE a developer can just color segments of the source code to assign them to a feature. Depending on the needed granularity, he can color classes, methods, statements, or even even any element in the source's *Abstract Syntax Tree (AST)* representation like a single parameter.

```
1 public class TxnManager {
2   void unRegisterTxn(Txn txn) throws DatabaseException {
3     allTxnLatch.acquire();
4     allTxns.remove(txn);
5     getMemoryBudget().updateMiscMemoryUsage(txn.getInMemorySize());
6     allTxnLatch.release();
7   }
8 }
```

Figure 9.1: Colored Source Code.

Contrasting conventional approaches with feature modules, the developer does not remove the feature code from the base code and adds it to a feature, but the IDE itself provides means to handle and compose the source. The IDE can hide certain features, or hide all code but certain features and thus allows multiple views on the project and its features. For example, a developer can decide to see only the code for feature *X*, or he can decide to see the full source code except features *X*, *Y* and *Z*. To create a configuration the user selects the features from a feature model and the IDE creates the resulting source code with the selected features.

The approach has some disadvantages, e.g., it does not support homogeneous extensions, or a feature is no longer coherently encapsulated by a single code artifact but an IDE is required to create a view. On the other hand, it solves almost every problem found in this thesis:

- **Laborious and error-prone Refactoring.** There is no need for refactoring in ColoredIDE, the developer directly colors the original code to assign it to features. To mark a code segment and assign it to a color has the same effect as the refactorings in this thesis. The back-end of ColoredIDE can be extended to still create AspectJ- or AHEAD-like features if certain restriction are introduced, but we consider this as an unnecessary step only needed for backward compatibility or easier transition for existing FOP or AOP projects.

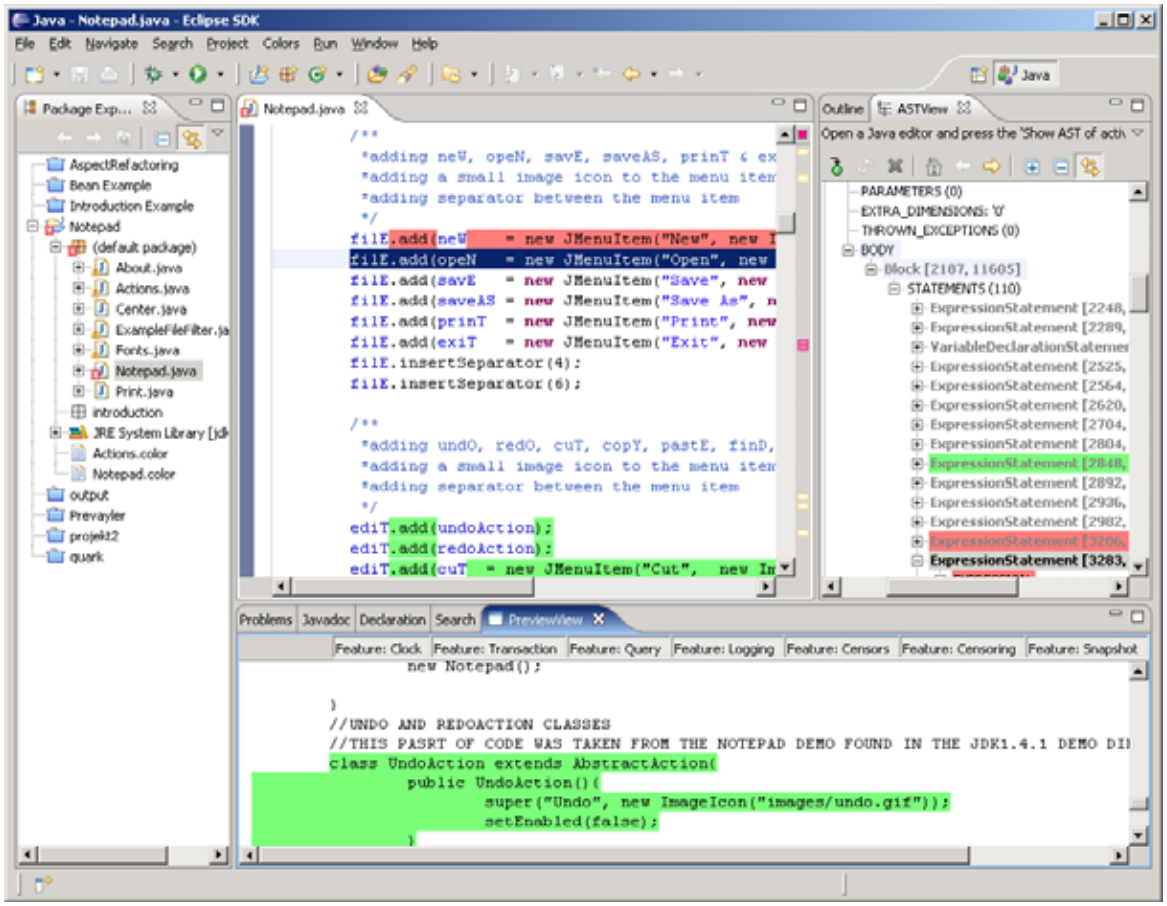


Figure 9.2: ColoredIDE Screenshot.

- **Finding Feature Code.** In our refactorings we removed feature code until all compilation errors were resolved to find all code that belongs to a feature. The process to find the full code base of a feature is similar in ColoredIDE: once a method is colored, the IDE reports errors for every call to this method that is not colored as well. There are various further tests that mimic the compilation process and thus helps in finding connected code quickly. Additionally, ColoredIDE can offer tool support to color all calls or find methods that are only called from a certain feature to move them to the feature as well. Early tests have shown that this works fast and conveniently.

- **Verifying Correctness.** ColoredIDE only allows to color elements of the AST, therefore the removing of colored code is safe because it only renders the AST without the colored nodes. To ensure compilable code without missing references or methods, dual tests check for the same colors of method definition and method call, types and their references, variable definition and variable reference, etc. Thus, ColoredIDE can ensure *syntactic correctness* for all configurations when the original code is compilable and all color checks are fulfilled.

- **Refactoring Limitations.** The Statement Extensions Problem, the Parameter Passing Problem, the Exception Introduction Problem, the Local Variables Access Problem and Scope Problems as observed in this thesis do not occur in ColoredIDE. The colored code remains at its original position, therefore it has full access to parameters, local variables and all methods in scope. Furthermore, arbitrary nodes of the AST can be colored, including statements inside a method, declared exceptions, or even parameters in method definitions or calls. There is no need for preliminary introductions of context objects or the like.

- **Understandability.** The developer does not need to learn a new language. He always can see the fully composed source code, or just a view with the features he is interested in. The coloring of code is as easy as selecting the target code and selecting the color it should have. An understanding of the AST below might help but will not be necessary in most cases. The selection of views and the creation of configurations is done with few IDE commands and simple toggle buttons.

- **Maintainability.** ColoredIDE avoids the error-prone constructs that modify or extend existing code. The developer can always see the fully composed source code, which makes maintenance easier than different code artifacts he has to compose in his head when he makes changes. Still the developer can hide features which are not affected from the maintenance work and thus reduces complexity.

- **Feature Optionality Problem.** The color model of ColoredIDE can be formally mapped to the derivative model of Liu et al. (2006). However the developer does not need to create derivative modules, he justs colors derivative code with the colors of both features and the IDE internally calculates the correct composition.

- **Feature Order.** An ordering of features is not required for ColoredIDE, because features are not applied as functions but have their fixed position in the source code. Because ColoredIDE has no own mechanism to quantify over join points the logical problems and accidental weaving cannot occur.

The key strength of ColoredIDE lies in the fact that it hides complexity. It frees the developer from the need to create feature directories, to describe complex transformations, or to create derivatives. On the other hand it gives him the most flexibility possible by allowing to color individual nodes of the AST.

Further research is required to analyze the limits and advantages of ColoredIDE on a larger projects. The ColoredIDE approach can furthermore be extended as a frontend to existing languages like AspectJ or AHEAD. Finally, it is possible to try to adapt ColoredIDE which is developed to decompose legacy applications as a general purpose SPL tool that can also be used to develop new projects.

Appendix A

The Dimension Model

To approximate the number of necessary derivatives features for further refactoring in Berkeley DB, we introduce a dimension model. Dimensions are groups of features that have simple semantic dependencies but that frequently crosscut the whole application and create technical dependencies by interacting with features from other dimensions. When a new feature is added to a dimension it is likely to interact with features from the other dimensions.

In Berkeley DB we have identified four dimensions: (1) synchronization and transactions, (2) memory and caching, (3) external information, and (4) database operations. Each of these dimensions forms a closed subsystem but cuts across the whole application:

- The synchronization and transaction dimension (features ATOMICTRANSACTIONS, LOCKING, LATCHES) inserts synchronization code almost in all parts of the application. It affects every database operation, most parts of internal memory management and caching, and statistics are collected about it.

- The memory and caching subsystem (incl. features MEMORYBUDGET, EVICTOR, LOOKAHEADCACHE, FILEHANDLECACHE) collects the overall memory usage all over the application, including the memory usage off transactions and individual operations. It is synchronized, logged and statistics are collected about it. Though individual caches do not interact with each other, they are applied all over the application in disjoint places.

- The external information dimension covers the logging and statistics features that collect information about the application's state and operations. For example, statistics are collected about many aspects of the memory and caching system and the transaction system. The logging system traces information about method execution from almost all parts of the application, including database operations, locking, and memory management.

- Database operations—although this dimension is currently represented only by the two features DELETEDBOPERATION and TRUNCATEDBOPERATION—also crosscut all other parts of the application. All database operations in Berkeley DB are

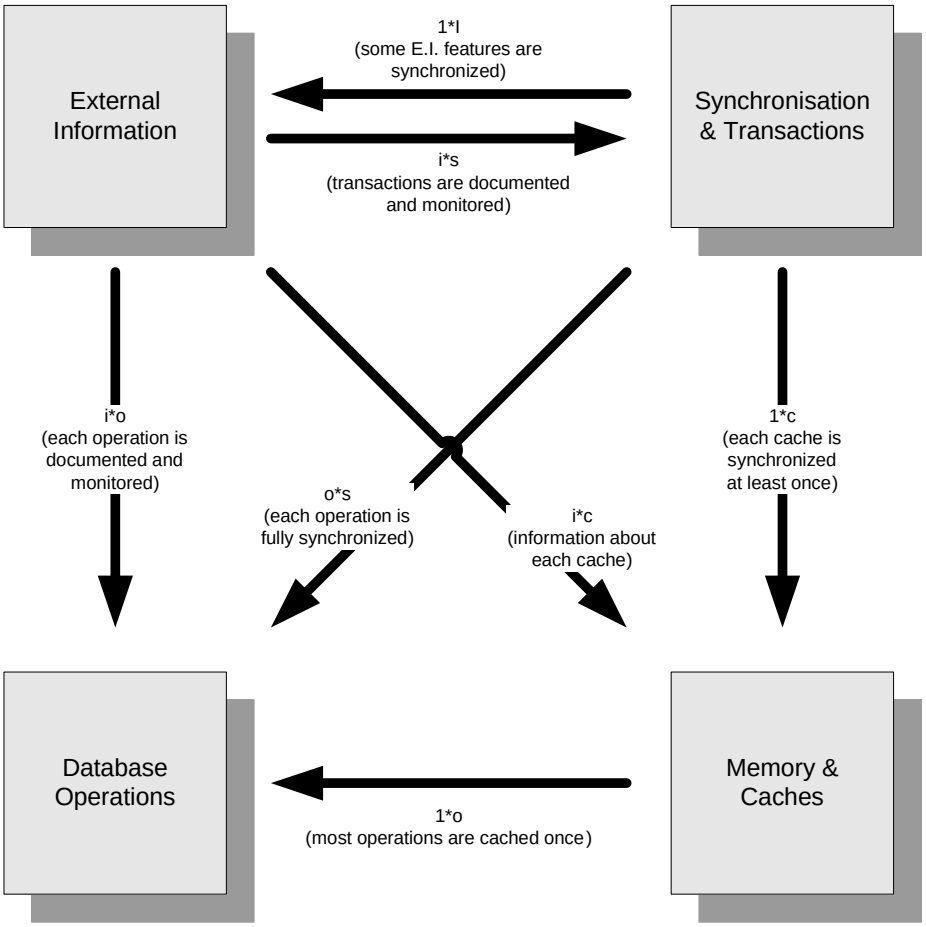


Figure A.1: Quantified Interaction between Dimensions.

thread safe and support transactions. Their memory consumption is carefully monitored and the "critical eviction" is called before certain operations. Finally, most database operations are logged. Consequently a new database operation feature will interact with most dimensions.

Some refactored features like CHECKSUMVALIDATION or SYNCHRONIZEDIO do not belong to any of these four dimensions. They are sometimes extended by one or more dimensions, but they do not extend any of them.

We can use these dimensions to approximate the number of interactions for further refactorings, based on the number of features in each dimension as defined in Figure A.2: Every new database operation feature approximately introduces new code that must be monitored by the memory dimension. Furthermore, each operation feature is presumably extended by most features from the synchronization dimension and at least by the logging feature from the external information dimension. Every cache in the memory and caching dimension will approximately synchronize at least one operation, but will be monitored by the external information dimension and synchronized by at least one feature from the

Name	Number of Features in Dimension
s	Synchronization and Transactions
c	Memory and Caching
i	External Information
o	Database Operations

Figure A.2: Variables of the Derivative Model.

synchronization dimension. Each feature from the external information dimension collects information about all dimensions usually on a very detailed level. Finally, the synchronization dimension synchronizes all other dimensions where necessary. The most important synchronization is done for the database operations, but also caches and external information collectors are synchronized where necessary.

We illustrate these interactions and the number of required derivative features in Figure A.1. Note, when two features mutually extend each other, both extensions can be implemented in one derivative feature. Therefore, the overall amount of features F is

$$F(s,c,i,o) = i \cdot o + i \cdot s + o \cdot s + i \cdot c + c + o$$

Using our current refactorings as values with $s = 3, c = 5, i = 2, o = 2$ we should expect 33 derivatives, relatively close to the actual amount of derivatives (42) we found .

When only one dimension is extended, the number of derivative features grows linearly because the derivative of F is constant:

$$\frac{\partial F(s,c,i,o)}{\partial s} = i + o$$

$$\frac{\partial F(s,c,i,o)}{\partial c} = i + 1$$

$$\frac{\partial F(s,c,i,o)}{\partial i} = s + c + o$$

$$\frac{\partial F(s,c,i,o)}{\partial o} = s + i + 1$$

With our current features we can thus, predict that each new synchronization feature will create 4 new derivatives, each caching feature 3, each external information feature 10, and each new database operation will create 6 new derivatives.

If n features are distributed over all dimensions the number of derivatives grows quadratically with n. The fastest growth is reached when the features are distributed equally over all four dimensions with $s = c = i = o = \frac{1}{4} \cdot n$:

$$F(\frac{n}{4}, \frac{n}{4}, \frac{n}{4}, \frac{n}{4}) = \frac{1}{4} \cdot n^2 + \frac{1}{2} \cdot n$$

It is necessary to emphasize that not all features belong to one of the dimensions. In Berkeley DB, only 12 of 38 features are part of any dimension. The features that are not in any dimension typically require no or only very few derivatives and are therefore not considered in this model. If we roughly estimate that one third of all features are in dimensions and these are equally distributed over all dimensions ($s = c = i = o = \frac{1}{4} \cdot \frac{n}{3}$), we get

$$F(\frac{n}{12}, \frac{n}{12}, \frac{n}{12}, \frac{n}{12}) = \frac{1}{36} \cdot n^2 + \frac{1}{6} \cdot n \approx 0.028 \cdot n^2 + 0.17 \cdot n$$

which is close to our empirical observations.

Note, the aim of this model is not to predict exact numbers but to give further insight on how the number of derivative features grows with larger feature models. This model confirms that the number grows quadratically but much slower than the maximum number of derivative features.

Bibliography

Jonathan Aldrich. Open Modules: Reconciling Extensibility and Information Hiding. In *SPLAT 04: AOSD Workshop on Software-engineering Properties of Languages for Aspect Technologies, Proceedings* (Lancaster, UK), 2004.

Jonathan Aldrich. Open Modules: Modular Reasoning About Advice. In Andrew Black, editor, *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming* (Glasgow, UK), volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer, 2005.

Michal Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange* (Vancouver, Canada), pages 67–72, New York, NY, USA, 2004. ACM Press.

Sven Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2007.

Sven Apel and Don Batory. When to Use Features and Aspects? A Case Study. In *GPCE '06: Proceedings of ACM SIGPLAN 5th International Conference on Generative Programming and Component Engineering* (Portland, OR, USA), 2006.

Sven Apel and Jia Liu. On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In *ADI '06: Proceedings of ECOOP Workshop on Aspects, Dependencies, and Interactions* (Nantes, France), 2006.

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In Robert Glück and Michael R. Lowry, editors, *Proceedings of the 4th International Conference on Generative Programming and Component Engineering* (Tallinn, Estonia), volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2005.

Sven Apel, Don Batory, and Marko Rosenmüller. On the Structure of Crosscutting Concerns: Using Aspects or Collaborations? In *AOPLE '06: Proceedings of GPCE Workshop on Aspect-Oriented Product Line Engineering* (Portland, OR, USA), 2006a.

Sven Apel, Christian Kästner, Thomas Leich, and Gunter Saake. Aspect Refinement. Technical Report 10, Department of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2006b.

Sven Apel, Thomas Leich, and Gunter Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceeding of the 28th international conference on Software engineering* (Shanghai, China), pages 122–131, New York, NY, USA, 2006c. ACM Press.

Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, 3880:135–173, 2006.

AspectJ-Team. *The AspectJ™ Programming Guide, Language Semantics Appendix*. Palo Alto Research Center, Inc, 2003. Accessed 2006-11-17.

Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development* (Chicago, IL, USA), pages 87–98, New York, NY, USA, 2005. ACM Press.

Don Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Edinburgh, UK), pages 702–703. IEEE Computer Society, 2004.

Don Batory. Feature Models, Grammars, and Propositional Formulas. In J. Henk Obbink and Klaus Pohl, editors, *SPLC 2005: Proceedings of the 9th International Software Product Lines Conference* (Rennes, France), volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

Don Batory. Extensible Cost Models and Query Optimization in GENESIS. *IEEE Database Engineering Bulletin*, 9(4):30–36, 1986.

Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In R. Laemmel, J. Saraiva, and J. Visser, editors, *Proc. Summer School on Generative and Transformation Techniques in Software Engineering* (Braga, Portugal), volume 4143 of *Lecture Notes in Computer Science*. Springer, 2006.

Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, 1992.

Don Batory, Jia Liu, and Jacob Neal Sarvela. Refinements and multi-dimensional separation of concerns. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering* (Helsinki, Finland), pages 48–57, New York, NY, USA, 2003. ACM Press.

Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, Boston, MA, USA, 2004.

Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: controlling the scope of change in Java. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (San Diego, CA, USA), pages 177–189, New York, NY, USA, 2005. ACM Press.

David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated Refactoring of Object Oriented Code into Aspects. In *ICSM 2005: Proceedings of the 21st IEEE International Conference on Software Maintenance* (Budapest, Hungary), pages 27–36. IEEE Computer Society, 2005.

David Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, 2006.

Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications* (Ottawa, Canada), pages 303–311, New York, NY, USA, 1990. ACM Press.

Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwe. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

Jeffrey R. Carter. Ada's design goals and object-oriented programming. *ACM SIGAda Ada Letters*, XIV(6):57–61, 1994.

Shigeru Chiba and Rei Ishikawa. Aspect-Oriented Programming Beyond Dependency Injection. In Andrew Black, editor, *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming* (Glasgow, UK), volume 3586 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2005.

Andy Clement, Adrian Colyer, and Mik Kersten. Aspect-Oriented Programming with AJDT. In *AAOS 2003: ECOOP Workshop on Analysis of Aspect-Oriented Software* (Darmstadt, Germany), 2003.

Curtis Clifton and Gary T. Leavens. Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy. In *SPLAT: Software engineering Properties of Languages for Aspect Technologies at AOSD '03* (Boston, MA, USA), 2003.

Yvonne Coady and Gregor Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In Mehmet Akşit, editor, *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development* (Boston, MA, USA), pages 50–59. ACM Press, 2003.

Leonardo Cole and Paulo Borba. Deriving refactorings for AspectJ. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development* (Chicago, IL, USA), pages 123–134, New York, NY, USA, 2005. ACM Press.

Adrian Colyer, Awais Rashid, and Gordon Blair. On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Computing Department, Lancaster University, Lancaster, UK, 2004.

Douglas Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

Brian de Alwis, Stephan Gudmundson, Greg Smolyn, and Gregor Kiczales. Coding issues in AspectJ. In *Proceedings of Workshop on Advanced Separation of Concerns in Object-Oriented Systems at the Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Minneapolis, MN, USA), 2000.

Merijn de Jonge and Joost Visser. Grammars as Feature Diagrams. In *ICSR7 Workshop on Generative Programming*, pages 23–24, 2002.

Marc Eaddy and Alfred Aho. Statement Annotations for Fine-Grained Advising. In *RAMSE '06: ECOOP'06 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, Proceedings* (Nantes, France), pages 89–100, 2006.

Ulrich Eibauer. Studying the Effects of Aspect Oriented Refactoring on Software Quality using HSQLDB as Example. Diplomarbeit, University of Passau, Passau, Germany, 2006.

Ulrich W. Eisenecker, Frank Blinn, and Krzysztof Czarnecki. A Solution to the Constructor-Problem of Mixin-Based Programming in C++. In *First Workshop on C++ Template Programming* (Erfurt, Germany), 2000.

Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Charleston, SC, USA), pages 270–282, New York, NY, USA, 2006. ACM Press.

Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming Is Quantification and Obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, MA, USA, 2005.

Florian Forster and Friedrich Steimann. AOP and the antinomy of the liar. In *FOAL '06: Workshop on the Foundations of Aspect-Oriented Languages at AOSD'06* (Bonn, Germany), pages 47–56, 2006.

Martin Fowler. *Refactoring. Improving the Design of Existing Code*. The Addison-Wesley object technology series. Addison-Wesley, Reading, MA, USA, 1999.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Irum Godil and Hans-Arno Jacobsen. Horizontal decomposition of Prevayler. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research* (Toranto, Canada), pages 83–100. IBM Press, 2005.

Martin L. Griss. Implementing product-line features by composing aspects. In *Proceedings of the first conference on Software product lines : experience and research directions* (Denver, CO, USA), pages 271–288, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Canada), pages 265–274. IEEE Computer Society, 2001.

Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (Boston, MA, USA), pages 60–69, New York, NY, USA, 2003. ACM Press.

Stefan Hanenberg, Christian Oberschulte, and Rainer Unland. Refactoring of Aspect-Oriented Software. In *Net.ObjectDays '03: Proceedings of the 4th International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World* (Erfurt, Germany), pages 19–35, 2003.

Jan Hannemann and Gregor Kiczales. Overcoming the Prevalent Decomposition in Legacy Code. In *Workshop on Advanced Separation of Concerns in Software Engineering at ICSE '01* (Toronto, Canada), 2001.

Jan Hannemann, Thomas Fritz, and Gail C. Murphy. Refactoring to aspects: an interactive approach. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange* (Anaheim, CA, USA), pages 74–78, New York, NY, USA, 2003. ACM Press.

Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development* (Chicago, IL, USA), pages 135–146, New York, NY, USA, 2005. ACM Press.

William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. *ACM SIGPLAN Notice*, 28(10):411–428, 1993.

Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *NODe '02: Proceedings of the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World* (Erfurt, Germany), volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2002.

Frank Hunleth and Ron K. Cytron. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems* (Berlin, Germany), pages 38–45. ACM Press, 2002.

Frank Hunleth, Ron Cytron, and Christopher Gill. Building Customizable Middleware using Aspect Oriented Programming. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA '01* (Tampa, FL, USA), 2001.

Masanori Iwamoto and Jianjun Zhao. Refactoring Aspect-Oriented Programs. In *The 4th AOSD Modeling With UML Workshop* (San Francisco, CA, USA), 2003.

K. Kang, S.Cohen, J.Hess, W.Novak, and S.Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA, USA, 1990.

Christian Kästner, Sven Apel, and Gunter Saake. Implementing Bounded Aspect Quantification in AspectJ. In *RAM-SE '06: ECOOP'06 Workshop on Reflection, AOP, and Meta-Data for Software Evolution, Proceedings* (Nantes, France), pages 111–122, 2006.

Gregor Kiczales and Mira Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In Andrew Black, editor, *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming* (Glasgow, UK), volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *ECOOP '01:*

Proceedings of the 15th European Conference on Object-Oriented Programming (Budapest, Hungary), volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

Christian Koppen and Maximilian Störzer. PCDiff: Attacking the Fragile Pointcut Problem. In *EIWAS: European Interactive Workshop on Aspects in Software* (Berlin, Germany), 2004.

Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. Tool support for feature-oriented software development: featureIDE: an Eclipse-based approach. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* (San Diego, CA, USA), pages 55–59, New York, NY, USA, 2005a. ACM Press.

Thomas Leich, Sven Apel, Marko Rosenmueller, and Gunter Saake. Handling Optional Features in Software Product Lines. In *Proceedings of OOPSLA Workshop on Managing Variabilities consistently in Design and Code* (San Diego, CA, USA), 2005b.

Thomas Leich, Sven Apel, and Gunter Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In Johann Eder, Hele-Mai Haav, Ahto Kalja, and Jaan Penjam, editors, *ADBIS '05: Proceedings of the 9th East European Conference on Advances in Databases and Information Systems* (Tallinn, Estonia), volume 3631 of *Lecture Notes in Computer Science*, pages 324–337. Springer, 2005c.

K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: an objective sense of style. In *OOPSLA '88: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (San Diego, CA, USA), pages 323–334, New York, NY, USA, 1988. ACM Press.

Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS99-01, Northeastern University, Boston, MA, USA, 1999.

Karl J. Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, 2003.

Jia Liu, Don Batory, and Srinivas Nedunuri. Modeling Interactions in Feature Oriented Software Designs. In Stephan Reiff-Marganiec and Mark Ryan, editors, *ICFI '05: Feature Interactions in Telecommunications and Software Systems VIII* (Leicester, UK), pages 178–197. IOS Press, 2005.

Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering* (Shanghai, China), pages 112–121, New York, NY, USA, 2006. ACM Press.

Roberto Lopez-Herrejon and Don Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE '01: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, pages 10–24, London, UK, 2001. Springer-Verlag.

Roberto Lopez-Herrejon, Don Batory, and William R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In Andrew Black, editor, *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming* (Glasgow, UK), volume 3586 of *Lecture Notes in Computer Science*, pages 169–194. Springer, 2005.

Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (Charleston, SC, USA), pages 68–77, New York, NY, USA, 2006. ACM Press.

Neil Loughran and Awais Rashid. Mining Aspects. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design at AOSD '02* (Enschede, The Netherlands), 2002.

Ole Lehrmann Madsen and Birger Moller-Pedersen. Virtual Classes: A powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications* (New Orleans, LA, USA), pages 397–406, New York, NY, USA, 1989. ACM Press.

Mike Mannion. Using First-Order Logic for Product Line Model Validation. In Gary J. Chastek, editor, *SPLC 2: Proceedings of the Second International Conference on Software Product Lines* (San Diego, CA, USA), volume 2379 of *Lecture Notes in Computer Science*, pages 176–187. Springer, 2002.

Hidehiko Masuhara and Kazunori Kawauchi. Dataflow Pointcut in Aspect-Oriented Programming. In Atsushi Ohori, editor, *APLAS: Proceedings of the First Asian Symposium on Programming Languages and Systems* (Beijing, China), volume 2895 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2003.

Nathan McEachen and Roger T. Alexander. Distributing classes with woven concerns: an exploration of potential fault scenarios. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development* (Chicago, IL, USA), pages 192–200, New York, NY, USA, 2005. ACM Press.

Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. *ACM SIGSOFT Software Engineering Notes*, 29(6):127–136, 2004.

Miguel P. Monteiro. Catalogue of Refactorings for AspectJ. Technical Report UM-DI-GECSD-200402, Escola de Engenharia, Universidade do Minho, Braga, Portugal, 2004.

Miguel P. Monteiro and João M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development* (Chicago, IL, USA), pages 111–122, New York, NY, USA, 2005. ACM Press.

Miguel P. Monteiro and João M. Fernandes. Object-to-Aspect Refactorings for Feature Extraction. Industry paper presented at *AOSD'2004*, Lancaster, UK, 2004.

Gail Murphy and Christa Schwanninger. Guest Editors' Introduction: Aspect-Oriented Programming. *IEEE Software*, 23(1):20–23, 2006.

Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: an exploratory study. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Canada), pages 275–284, Washington, DC, USA, 2001. IEEE Computer Society.

Alexander Nyssen, Shmuel Tyszberowicz, and Thomas Weiler. Are Aspects useful for Managing Variability in Software Product Lines? A Case Study. In *Aspects and Software Product Lines: An Early Aspects Workshop at SPLC-Europe 2005* (Rennes, France), 2005.

Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Vancouver, BC, Canada), pages 99–115, New York, NY, USA, 2004. ACM Press.

Oracle. *Oracle Berkeley DB Product Family Data Sheet.* Oracle Corporation, 2006a. `http://www.oracle.com/technology/products/berkeley-db/pdf/berkeley-db-family-datasheet.pdf`.

Oracle. *Oracle Berkeley DB Java Edition Data Sheet.* Oracle Corporation, 2006b. `http://www.oracle.com/database/docs/berkeley-db-je-datasheet.pdf`.

Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In Andrew Black, editor, *ECOOP '05: Proceedings of the 19th European Conference on Object-Oriented Programming* (Glasgow, UK), volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer, 2005.

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

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering* (Atlanta, GA, USA), pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

Markus Pizka. Straightening Spaghetti-Code with Refactoring? In Hamid R. Arabnia and Hassan Reza, editors, *SERP '04: Proceedings of the International Conference on Software Engineering Research and Practice* (Las Vegas, NV, USA), volume 2, pages 846–852. CSREA Press, 2004.

Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming* (Jyväskylä, Finland), volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.

Tobias Rho and Günter Kniesel. Uniform Genericity for Aspect Languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn, 2004.

Daniel Sabbah. Aspects: from promise to reality. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development* (Lancaster, UK), pages 1–2, New York, NY, USA, 2004. ACM Press.

Sleepycat. *Getting Started with Berkeley DB Java Edition*. Sleepycat Software, Inc, 2.1 edition, 2006.

Yannis Smaragdakis and Don Batory. Mixin-Based Programming in C++. In *GCSE '00: Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*, pages 163–177, London, UK, 2001. Springer-Verlag.

Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPLSA '86: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (Portland, OR, USA), pages 38–45, New York, NY, USA, 1986. ACM Press.

Friedrich Steimann. The paradoxical success of aspect-oriented programming. *ACM SIGPLAN Notices*, 41(10):481–497, 2006.

Maximilian Stoerzer and Juergen Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance* (Budapest, Hungary), pages 653–656, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (Lisbon, Portugal), pages 166–175, New York, NY, USA, 2005. ACM Press.

Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Angeles, CA, USA), pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

Aleksandra Tesanovic, Ke Sheng, and Jorgen Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *IDEAS '04: Proceedings of the International Database Engineering and Applications Symposium*, pages 291–301, Washington, DC, USA, 2004. IEEE Computer Society.

Lance Tokuda and Don Batory. Evolving Object-Oriented Designs with Refactorings. *Journal of Automated Software Engineering*, 8(1):89–120, 2001.

Charles Zhang and Hans-Arno Jacobsen. Resolving feature convolution in middleware systems. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Vancouver, Canada), pages 188–205, New York, NY, USA, 2004. ACM Press.

Charles Zhang and Hans-Arno. Jacobsen. Quantifying aspects in middleware platforms. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (Boston, MA, USA), pages 130–139, New York, NY, USA, 2003. ACM Press.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, February 7, 2007

Christian Kästner