A Case Study Implementing Features Using AspectJ

Christian Kästner
School of Computer Science
University of Magdeburg
39106 Magdeburg, Germany
ckaestne@ovgu.de

Sven Apel
Dept. of Informatics and Math.
University of Passau
94030 Passau, Germany
apel@uni-passau.de

Don Batory
Dept. of Computer Sciences
University of Texas at Austin
Austin, Texas 78712
batory@cs.utexas.edu

Abstract

Software product lines aim to create highly configurable programs from a set of features. Common belief and recent studies suggest that aspects are well-suited for implementing features. We evaluate the suitability of AspectJ with respect to this task by a case study that refactors the embedded database system Berkeley DB into 38 features. Contrary to our initial expectations, the results were not encouraging. As the number of aspects in a feature grows, there is a noticeable decrease in code readability and maintainability. Most of the unique and powerful features of AspectJ were not needed. We document where AspectJ is unsuitable for implementing features of refactored legacy applications and explain why.

1. Introduction

Aspect-oriented programming (AOP) is a paradigm aimed at modularizing crosscutting concerns by encapsulating replicated, scattered, and tangled code in *aspects* [30]. Features are used to distinguish programs within a *software product line (SPL)* [43, 9, 27], where a *feature* is an increment in functionality [9]. Studies have suggested that features of an SPL be implemented by aspects [22, 16, 47, 32, 26, 46, 14], as features often have an inherent crosscutting behavior to them.

In this paper, we present a case study on refactoring a legacy application into an SPL using aspects to implement features. Our study uses *Oracle Berkeley DB JE*[1]. Berkeley DB is an embedded database system (about 84 KLOC), released as an open source project. It can be embedded in other applications as a fast transactional storage engine. We chose Berkeley DB because of its size and because its domain is well-understood. Taking an extractive approach to SPL development [13], we refactored Berkeley DB into optional features, to be able to build tailored variants of it. For example, we might create a variant without any debugging code (e.g., logging, leak checks, B$^+$-tree verification, checksums) that is faster for production use. Our goal was to evaluate the suitability of AspectJ, the most popular AOP language, to implement features.

Prior studies recommended using aspects to implement features; in particular it was reported that source code quality and configurability were improved. However, our experience with Berkeley DB makes us skeptical. In this paper we explain and justify our finding that aspects written with AspectJ are not suitable for implementing features in feature-refactored legacy applications, I particularly when heterogeneous crosscuts (the predominant form of features), code readability, fragility, and coupling are considered. We assume minimal knowledge of AspectJ and AOP in this paper.

2. Perspective

We started this project with a basic knowledge of AspectJ. We learned from examples given in Laddad's book [31] and various papers. We even built extensions to an AspectJ compiler [4, 29]. AspectJ provided all the constructs needed to implement features, including impressive new possibilities like pattern expressions and conditional pointcuts. However up to this point, we had not *used* AspectJ *in practice* to implement a larger program that exceeded the small illustrative examples that dominate the literature. This and prior studies encouraged us to tackle a larger case study, initially only to analyze the concept of functional weaving for another branch of research on aspects [6, 29].

However, we encountered problems early on in our refactoring. We anticipated difficulties, but their severity was unexpected. Studying the suitability of AspectJ for implementing features became a more important part of the case study than the initial idea of functional weaving.

We believe that the problems we encountered will be familiar to others who have used AspectJ extensively. While we are well-aware that *any* language can be abused and statements stemming from inappropriate usage that claim

[1]`http://oracle.com/technology/products/berkeley-db`

Feature	LOC	EX	AT	Description
AtomicTransact.	715	84	19	Part of the transaction system that is responsible for atomicity.
CheckpointerD.	110	14	4	Daemon to create checkpoints in the log.
ChecksumValid.	324	32	8	Checksum read and write validation of persistence subsystem.
ChunkedNIO	52	2	1	Chunked new I/O implementations.
CleanerDaemon	129	9	2	Daemon to clean old log files.
CriticalEviction	63	10	7	Evictor calls before critical operations to ensure enough memory.
CpBytesConfig	41	7	5	Configuration options for the Checkpointer by size.
CpTimeConfig	59	6	5	Configuration options by time.
DbVerifier	391	16	10	Debug facility to verify the integrity of the B^+-tree.
DeleteDbOp.	226	31	13	Operation to delete a database.
DirectNIO	6	1	1	Direct I/O access.
DiskFullErrorH.	41	4	2	Emergency operations on a full disc error.
EnvironmentLock	61	5	2	Prevents two instances on the same database directory.
Evictor	371	20	9	Subsystem that evicts objects from cache for the garbage collector.
EvictorDaemon	71	9	3	Daemon thread that runs the Evictor when a memory limit is reached.
FileHandleCache	101	6	2	File handle cache.
FSync	130	5	1	File synchronization for writing log files.
INCompressor	425	21	4	Removes deleted nodes from the internal B^+-tree.
IO	38	2	1	Classic I/O implementation.
Latches	1835	155	28	Fine grained thread synchronization.
LeakChecking	43	4	2	Debug checks for leaking transactions.
Logging	1115	132	24	Debug logging facilities, separated in 10 features for different logging levels and handlers, not listed here.
LookAheadCache	84	6	2	Look ahead cache for read operations.
MemoryBudget	958	118	28	Observes the overall memory usage.
NIO	26	2	1	New I/O implementation.
Statistics	1867	345	30	Collects runtime statistics like buffer hit ratio throughout the system.
SynchronizedIO	26	2	1	Synchronized I/O access.
TreeVisitor	138	24	9	Provides a Visitor to traverse the internal B^+-tree.
TruncateDbOp.	131	5	3	Operation to truncate a database.

AD - Pieces of advice; EX - Extensions (advice, introductions);
AT - Number of types affected by the feature.

Table 1. Refactored features of Berkeley DB.

language unsuitability will be misleading, readers with minimal background with AspectJ will recognize many of the issues that we raise. To make our scientific investigations above-board, we make the source code of our case study available for all to inspect[2]. We hope that our work, and confirmations of our results, can serve as a starting point to improve programming languages and support developers in selecting the right language for the right problem in SPL development.

3. Refactoring Berkeley DB

Refactoring a legacy application into features is known as *feature-oriented refactoring* [34] (a.k.a. *horizontal decomposition* [47]). Berkeley DB is written entirely in Java and can be included into an application as a library. Its performance and transaction safety make it popular in open source and commercial applications. It consists of five large subsystems as shown in Figure 1: access methods provide an abstraction and API to the user, a B^+-tree for internal data storage, diverse caching and buffering mechanisms, a concurrency and transaction system, and finally a persistence layer. It is possible to deactivate some parts like transactions at startup, but it is not possible to create a tailored variant of the database system during build time that excludes unnecessary code.

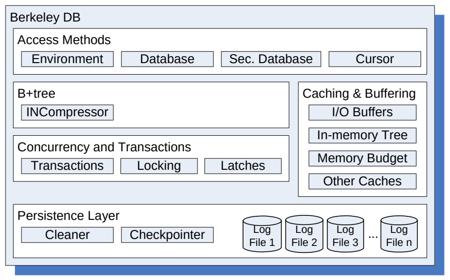

Figure 1. Overview of Berkeley DB.

By analyzing the domain, manual, configuration parameters, and source code, we identified many parts of Berkeley DB that represent increments in program functionality that were candidates to be refactored into features. These features were implicit in the legacy code. They varied from small caches, to entire transaction or persistence subsystems [28]. All identified features represent program functionality as a user would select or deselect them when customizing a database system. From these features, we chose 38 for actual refactoring. They modularized code from all parts of Berkeley DB and varied in size. A partial list with a short description and the size for each feature is given in Table 1[3]. A feature diagram that shows how these features relate is depicted in Figure 2. Lines indicate the hierarchy between features and subfeatures, while empty circles represent optional features and filled circles required ones. The additional line between the connections of IO and NIO represent alternatives, only one of these features can be selected at a time.

With these features, we have an SPL. We can build different tailored variants of Berkeley DB by selecting which features to include for compilation, e.g., log severe level events and exclude checksum validation. Even though there are dependencies between these features [28], we can create thousands of different variants of Berkeley DB.

Infrastructure. Although it is possible to implement large features in one aspect and even introduce classes as inner classes, it became necessary to decompose large aspects during development to keep them readable as suggested

[2] `http://wwwiti.cs.uni-magdeburg.de/iti_db/berkeley/`

[3] All statistics in this paper were collected with two self-written tools *ajdtstats* and *abcstats* which collect information from the internal structures from the AspectJ compilers *ajc* and *abc*. The relevant parts of these tools are published together with the source code of the study.

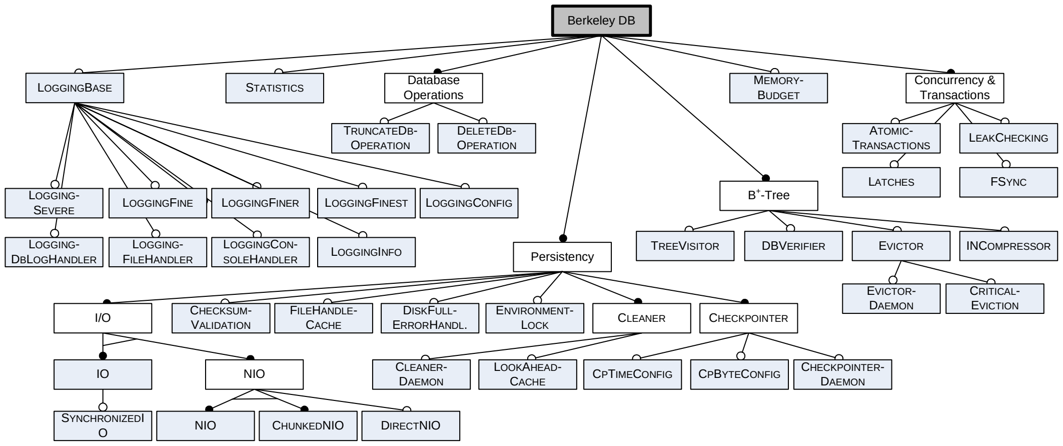

Figure 2. Feature Diagram.

in [5, 3, 2]. Unfortunately, AspectJ provides no infrastructure to define features, e.g., to group multiple cooperating aspects and classes together [36]. Further, the only way to compose an application with features is to manipulate the compiler's build path, to include or exclude certain aspect files.

It was thus necessary to impose an external infrastructure on the project. We used AHEAD [9] in which each feature is represented by a distinct directory ('containment hierarchy') that can contain multiple code artifacts. This way we used the AHEAD Tool Suite to describe features and dependencies and to select configurations with a graphical feature modeling tool [8].

The Refactoring Process. To refactor features from the original code into aspects, we used various OOP-to-AOP refactorings as suggested in recent research (e.g., [24, 39, 15, 11]). When possible, we moved whole classes or interfaces to a feature directory, or used the *Extract Introduction* refactoring [24, 39] to move methods or fields to aspects. Furthermore, we had to create advice to extend join points throughout the code. Common were *Extract Beginning* and *Extract End* refactorings [11, 24, 15] to move code from the beginning or the end of a method to an aspect. In cases where we had to refactor statements in the middle of a method we used either *Extract Before/After Call* refactorings [11], or prepared code with an object-oriented *Extract Method* refactoring [21]. If neither were possible, we created *hook methods*. Hook methods are empty methods placed in the base code for later extension. They have no purpose other than to provide a join point inside a method that can be extended by an aspect. As hook methods are considered 'unnatural' by AOP researchers [40], we used them only as a last resort. Such refactored extensions are *heterogeneous*, i.e., they affect only one join point each. Where possible, we created *homogeneous* advice, i.e., advice that extends multiple join points, by combining two refactored advice statements with the *Extract Pointcut* refactoring [15].

The Extract Introduction refactoring was used the most, followed by the Extract Beginning/End refactorings. Unfortunately, hook methods were often needed. In Table 2 we list each refactoring and how often it was used.

Refactoring	# times used
Extract Introduction (Method)	365
Extract Beginning/End	214
Extract Introduction (Field)	213
Create Hook Method	164
Extract Before/After Call	121
Move Class to Feature	58
Extract Method	15
Move Interface to Feature	4

Table 2. Refactorings used in Berkeley DB.

Even though we used refactorings described in prior publications, we performed all of them manually as we found no tool that could be used productively. We refactored one feature after another into one or multiple (up to 45) aspects per feature. Of our 38 refactored features, 16 were small with less than 140 LOC and 10 or fewer refactorings. The features LATCHES, STATISTICS, LOGGING, and MEMORY-BUDGET were large with 958–1864 LOC and required between 118 and 345 refactorings. These features affected between 24 and 30 classes each. All other features have a size in-between. Due to technical difficulties described below which we already observed in medium-sized features, we could not refactor very large features, like the whole persistence subsystem to create an in-memory database. With these 38 features, we refactored about 10 % of the code base of Berkeley DB. Afterwards, we stopped the refactoring process because it became repetitious and we did not expect further insights.

```
 1 public class IN {
 2   public int insertEntry1(CR entry) { //...
 3     if (nEntries < entryTargets.length) { //...
 4       updateMemorySize(0, getInMemorySize(index));
 5       adjustCursorsForInsert(index); //...
 6     }
 7   }
```

```
 8 public aspect MemoryBudget {
 9   before(IN in, int index):
10     call(void IN.adjustCursorsForInsert(int)) &&
          this(in) && args(index) &&
          withincode(int IN.insertEntry1(CR)) {
11     in.updateMemorySize(0, in.getInMemorySize(index));
12   }
13 }
```

Figure 3. *Extract Before Call* **Refactoring.**

```
pointcut latchedTxnMethods(TxnManager mgr): this(mgr) &&
 (execution(void TxnManager.registerTxn(Txn)) ||
  execution(void TxnManager.unRegisterTxn(Txn)) ||
  execution(long TxnManager.getFirstActiveLsn()) ||
  execution(Stats TxnManager.txnStat(StatsConfig)));
```

Figure 4. Enumerated homog. pointcut.

4. Observations

We review our observations during the refactoring and of the resulting feature code in this section. First, we identify the AspectJ language constructs that we used and the limitations we observed in using them. Then, we explain why we feel that the resulting feature code is hard to understand and hard to maintain. Finally, we comment on the utility of existing AspectJ tools that could be used for refactoring.

4.1. Used Language Constructs

AspectJ has an expressive language. Constructs range from static introductions, several types of advice, and up to 25 different pointcut designators. We found that we needed only basic designators to implement features.

The majority of all extensions (57 %) were static introductions. Overall, we introduced 4 interfaces, 58 classes, 365 methods, and 213 fields. To introduce methods and fields, we used AspectJ's inter-type declarations.

The next most frequently-used extension was simple method extensions, i.e., extensions to whole methods, implemented with advice at *execution* join points. Method extensions were used 214 times.

Similarly often, we needed extensions to statements or sequences of statements inside a method like Line 4 in Figure 3. As such extensions are not supported by AspectJ because it does not provide statement level join points, we had to use either (1) *call* join points to approximate statement extensions, typically with a combination of *call* and *withincode* pointcuts (cf. Fig. 3, L. 8–13), (2) extract the statements to a new method, or (3) introduce calls to empty methods as hooks for later extension. We tried to avoid hook methods where possible, but we could only emulate statement extensions with other pointcuts in 54 cases and had to create 164 hook methods.

A surprising result for us is that language constructs unique to AspectJ were rarely used. Conditional pointcuts like *if*, *this*, or *cflow* were used only in rare cases to experiment with workarounds for limitations discussed later.

But more importantly, homogeneous extensions were rarely used as shown in Table 3. Of 482 advice statements, only 7 used pattern expressions in their pointcuts to extend more than one join point. More often, multiple join points were matched explicitly, but they were enumerated with the || operator in one pointcut (cf. Fig. 4). This allowed us to avoid declaring the same advice body two or three times, and rarely more than three times. Homogeneous statement extensions, i.e., *call*, *set* or *get* pointcuts that match multiple join points were infrequently used as well. Overall, less than 11 % of all advice is homogeneous, and only 2 % extends more than 3 join points. These numbers are in line with experience from our previous studies [3, 2, 35].

	# extended 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 3. Homogeneous dynamic extensions.

4.2. Language Limitations

Various limitations prevented us from implementing features in a concise way. We often used workarounds which caused complex or 'strange' designs. Due to the restricted space we cannot discuss all observed limitations and some are probably even very specific to Berkeley DB. Instead we focus only on four that occurred frequently.

Statement Extension Problem. As stated above, we frequently required extensions to a statement or a sequence of statements inside a method. In some cases, we were able to emulate this by extending a *call* join point, but in many other cases we had to create hook methods as described in [40]. On the one hand, advice to emulate statement extensions requires fairly complex and fragile pointcuts, on the other hand, hook methods give a 'strange' look to the base code.

Local Variables Access Problem. Another problem related to statement extensions is that with AspectJ we were

```
 1  public class Tree {
 2    public long insert(LeafNode ln, byte[] key, ...) {
 3      BottomNode bin = findBINForInsert(key, ...);
 4      long position = ln.log(key, ...);
 5      bin.updateEntry(ln, position, key);
 6      bin.clearKnownDeleted();
 7      trace(bin, ln, position);
 8      ...
 9    }
10  }
```

```
11  public class Tree {
12    public long insert(LeafNode ln, byte[] key, ...) { ...
13      bin.clearKnownDeleted();
14      hook(bin, ln, position);
15      ...
16    }
17    void hook(BottomNode b, LeafNode l, long p) {}
18  }
19  public aspect TreeLogging {
20    before(BottomNode bin, LeafNode ln, long pos):
21      execution(void Tree.hook(..)) && args(bin,ln,pos) {
22      trace(bin, ln, pos)
23    }
24  }
```

Figure 5. Local Variables Access Problem.

not able to access local variables of a method. For example, the variables *bin* and *position* in the simplified excerpt in Figure 5 (L. 1–10) are inaccessible when refactoring the *trace* method. This forces us to either reproduce (copy) program logic in advice, to create method objects [21, p. 135f], or to create a hook method that exposes local variables as parameters (Fig. 5, L. 11–24).

Exception Introduction Problem. AspectJ cannot alter a method's signature, which also means that features cannot add new exceptions to existing methods. For example, to synchronize code with our LATCHES feature, we wanted to use around advice to acquire a latch before and release it after an operation. Unfortunately, the *acquire* operation can throw an exception when the latch is already held. Therefore, we can extend only methods that already declare this exception in the base code. For example we cannot synchronize a standard Java list, e.g., *'around() : call(boolean List.add(Object)) && within(Database)'*, because its operations do not declare any concurrency exception.

This forced us to either use runtime exceptions, or to declare all throwable exceptions of all features in the base code and to create hook methods to encapsulate API calls that do not declare the required exceptions.

Scope Problem. In Berkeley DB, the scope modifiers (*private*, *protected*, *public*) were used for a carefully designed concept that allows users to access only few interface methods. Unfortunately, scope problems during refactorings forced us to frequently change the scope modifiers. For example, AspectJ does not support *protected* inter-type declarations. Furthermore, even declaring an aspect as privileged does not allow complete access, e.g., it is not possible to access non-public classes or inner classes. We frequently had to increase the scope of methods, fields, or classes, thus destroying the scope concept of the Berkeley DB design.

4.3. Coupling and Pointcut Fragility

Our aspects were fragile and hard to maintain. Especially in later development steps this made it surprisingly difficult to refactor more features without breaking existing ones. There are several reasons.

Advice applies extensions to join points selected by pointcuts. The join points themselves show no sign that they are extended, i.e., coupling is implicit. Our statistics tool counted 21 795 join points in Berkeley DB, but only 528 (2.4 %) were extended[4]. To find out which join points are extended it is necessary to understand all aspects or use tool support.

Further, we observed that our features heavily depend on implementation details and are strongly coupled. A feature like STATISTICS not only relies on public methods, but it also extends join points throughout the application on a fine grained level to collect information. Of our 151 aspects, we had to declare 127 as privileged because they needed access to such details. If we took the advice often given in the AOP literature to rely only on interface methods that are not likely to change (e.g., in [31]), we could rarely implement any of our features, or we had to introduce many explicit public extension points like hook methods.

This strong and implicit coupling makes maintenance and evolution harder. It is not possible to make any local changes without understanding *all* aspects or without a tool that warns about extensions to a certain piece of code.

This problem is intensified by the fact that AspectJ's pointcuts are fragile [45, 23]. Changes to the source code, like renaming a method or moving code from the base to an aspect, can change join points. When pointcuts are not updated accordingly, the set of extended join points can change silently, thus changing the program behavior. Heterogeneous extensions that are not applied any more create a compiler warning, but homogeneous extensions, where the set of join points changes, signal nothing. To use pattern expressions like *'execution(* Database.put*(..))'* also for heterogeneous extensions might create an illusion of stability against changes, however, in our experience this makes it even harder to detect exceptions and fosters accidental weaving [37].

We observed such silent changes during development. We refactored features sequentially, and only when we checked all features in a later step did we discover that some pointcuts matched an incorrect set of join points, because they were not

[4]Some AOP publications distinguish between 'join points' which occur dynamically during the execution of the program and 'join point shadows', the location of a join point in the static source code. Our statistics refer to the latter.

updated accordingly. For example, LOGGING and LATCHES broke frequently because some code they logged or synchronized was moved. In the beginning with few features it was still possible to keep track of all pointcuts, but as the number of aspects and the number of pointcuts in those aspects increased, changes to join points which required updating a pointcut occurred more often, and we missed some of them. In our experience the most frequent causes 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 like *Extract Method* we made to create new join points, and (3) moving code from a method to advice and thus either moving or removing join points.

4.4. Readability and Understandability

The syntax of AspectJ can be hard to read and understand. Small examples in books and papers are illustrative and easy to understand in isolation, but when we actually implemented large features and wanted to show them to others we recognized several problems.

Increased Code Size. Simple extensions such as method extensions require overly complex constructs. In Figure 6 we show two ways to write a method extension that calls a logging method after each execution of the *put* method. The first version is a method extension as it would be written in object-oriented frameworks or collaboration-based designs; the second is an AspectJ solution of the same extension. The AspectJ solution is about twice as long and more complex: it needs 4 pointcut designators: *execution*, *args*, *within*[5], and *this*. Furthermore the parameters are repeated five times: in the pointcut declaration, in the execution pointcut designator, in the *args* pointcut designator, in the advice declaration, and in the pointcut reference. By using anonymous pointcuts, it is possible to shorten the construct but it is still larger than the alternative version.

```
public void delete(Transaction txn, DbEntry key) {
  super.delete(txn, key);
  Tracer.trace(Level.FINE, "Db.delete", this, txn, key);
}
```

```
pointcut traceDel(Database db, Transaction txn,
    DbEntry key) :
  execution(void Database.delete(Transaction, DbEntry))
  && args(txn, key) && within(Database) && this(db);
after(Database db, Transaction txn, DbEntry key):
  traceDel(db, txn, key) {
  Tracer.trace(Level.FINE, "Db.delete", db, txn, key);
}
```

Figure 6. Syntax comparison.

[5]The *within* pointcut designator is necessary to not affect subclasses of *Database*.

We measured the overall pointcut complexity, by counting how many pointcut designators were used per advice after resolving named pointcuts. Only 5 pointcuts (1 %) used just one designator, 70 pointcuts (14 %) used two. The large majority used three (194; 40 %) or four (176; 36 %) designators, even though most of them are just simple method extensions. Pointcuts with more designators were rare (9 %) and were often caused by explicit enumerations or *cflow* designators.

Third-Person Perspective. AspectJ uses the *third-person perspective*: advice 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.

We frequently required access to the extended object in our refactorings. 400 advice declarations—83 % of all—used a *this* or *target* pointcut designator to capture it.

The necessity to access an extended object with a pointcut makes not only the advice declaration larger, it makes also the extension harder to read. In Java, the *this* keyword is optional and it is common practice not to use it, unless required. Usually one expects 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 potential increases because methods introduced with inter-type declarations are written in a first-person perspective, thus both perspectives are frequently mixed in one aspect.

Advanced Extensions. Homogeneous pointcuts, unless they use very simple pattern expressions, often require tool support to find all matched join points. Further, 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 work. For example, the pointcut in Figure 7 is a shortened version of the one in Figure 6. We observed this practice in other AspectJ projects as well (cf. [2]). However, in our perception, 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 whether it is homogeneous or whether the pattern was just used for convenience.

```
pointcut traceDel(Database db, Transaction txn,
    DbEntry key) :
  execution(* delete(..)) && args(txn, key) && this(db);
```

Figure 7. Simplified heterogeneous pointcut.

Scaling Aspects. In Berkeley DB we found many medium-sized and large features with over 100 LOC and over 10 advice statements or introductions. For example, the LATCHES feature, one of our larger features, has 104 pieces of advice and 51 inter-type declarations in 1835 LOC. Small AspectJ examples and our small features are usually easy to understand. However, to read and understand a feature with 104 pieces of nontrivial advice, mixed together in one or more aspects, is harder. Beyond a certain feature size, features are not understandable without tool support.

4.5. Tool Support

A frequent suggestion to avoid some of the problems mentioned above is to use tools. For example, to solve the implicit coupling that prevents local changes without understanding all aspects or the decreased readability of extensions with pattern expressions, tools can visualize crosscuts; to cope with pointcut fragility, tools to compare matched join point sets are developed.

Such an IDE used during our development was Eclipse with the AJDT plug-in. This IDE helped us a lot by showing markers at the source code where advice applies, and by showing which join points an advice statement extends. Even though this tool is good, it has three drawbacks.

First, it can show markers and links only on fully compilable source code. When we refactored large features, we frequently had syntax errors in our source code for multiple development steps until we could compile the completely refactored feature again. During this time we did not have any markers that may warn us not to change certain join points.

Second, to update markers after source code changes requires rebuilding the whole application (incremental building is possible but not reliable). As rebuilds of Berkeley DB took about one minute, we did not perform them often because they severely interrupted the development process.

Finally, markers are only shown for aspects that are included in the current configuration. The used tools are not designed for SPLs, we just used them for this purpose. This forced us to always develop the application with all features included. In our project this only slowed down the compilation process, however when dealing with alternative, mutual exclusive features, this might pose a significant problem.

5. Discussion and Remedies

In this section, we discuss the insights gained from our observations and possible remedies to the problems we identified previously. We focus on three points: (1) the suitability of AspectJ for implementing features compared to other approaches, (2) the role of tool support in AOP, and (3) inherent feature complexity.

5.1. Right Tool for the Right Task

With few exceptions, we used only basic extension mechanisms like introductions, method extensions, and statement extensions. Advanced mechanisms that are unique to AspectJ like conditional extensions or homogeneous crosscuts were rarely needed. In those few cases, the third-level perspective is required and by the use of pattern expressions it is possible to actually decrease code size and increase quality metrics as shown in various studies. However, for the vast majority of basic heterogeneous extensions that are typical of features, the use of AspectJ seems counterproductive.

There are possible solutions to many of the mentioned limitations and problems. Most require extensions to the Java or AspectJ language, like different coupling mechanisms [1], statement annotations [18], or less fragile pointcut languages [23]. Others suggest new tools to counter problems, like the *PCDiff* tool to detect changed pointcuts [45], or aspect-aware refactorings to safely evolve pointcuts [24, 39]. However, we think that these suggestions make the language even more complex and even harder to use, where most of the time only a few basic extensions are needed. They only treat symptoms and do not address the cause, which is using a language for a purpose for which it is not designed.

In SPLs, where aspects are needed to encapsulate advanced crosscutting concerns, an integration of aspects into other languages that were specifically designed for OO collaborations seems favorable. Several such approaches have been proposed recently [5, 2, 7, 33].

5.2. Tool Support

The emphasis on tool support in AOP, e.g., to solve problems of fragility, implicit coupling, and hard to understand pointcuts, only treats symptoms but not the cause of the problems.

Progress in software engineering has broken periodically with principles of locality multiple times. First subroutines and later object-oriented programming structured programs in novel ways. Understanding the source code became harder. For example, Carter reflects on the design of *ADA* 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 [12]. The development of modern large-scale object-oriented applications also often requires tool support to navigate the source code, e.g., to display a class hierarchy or all calls to a method. Constructs like design patterns, or J2EE beans make the development very abstract and tools crucial for an understanding. History has shown that most developers do not mind the acclaimed 'reduced readability' and the necessary tool support, for the improvements gained.

Nevertheless, we still think that there is a significant difference between on the one hand having to look up a superclass or calls, and on the other hand searching the *whole* source code for affected join points. Therefore, developers must demand much higher standards for AOP tools because they depend on them instead of just using them for support. In our experience, understanding or maintaining aspect-oriented code in Berkeley DB without tool support is impossible because of the implicit coupling and fragility. The existing tool problems, like only providing information after a complete and successful build, are not acceptable for a production setting.

When using AspectJ for optional features, the whole tool support must be adjusted. It is not possible to merely use the existing AspectJ tools for SPLs, instead tools must be built to support optional features. As stated above, it is not acceptable to compile the project always with all (even alternative) features included, just to get source code markers to work around fragile pointcuts.

Basically, there are two different solutions for this problem. We can either (1) use language mechanisms that does not require such high degree of tool support (e.g., collaborations [44, 9]), or (2) we can use a tool driven approach, where the tool is an essential part. This would limit the extension mechanism to the capabilities provided by that specific tool, but we could rely on this tool. AspectJ stands between those two solutions, it requires a tool but a tool is not part of the solution but just developed as an add-on.

5.3. Inherent vs. Added Complexity

Berkeley DB was not designed for feature-extensibility. Its features were closely integrated into the original design and internal implementation. It had a clean design, but feature-extensibility was not one of its design goals. We tried to decompose it instead of redesigning it. This is in line with prior studies on implementing features with AOP.

Some of the observed complexity is inherent to refactored features. In an application designed for extensibility, e.g., with a framework design, a feature extends only well defined extension points. In contrast, the original feature implementation in a legacy application is often interwoven strongly with the application's internal implementation. The refactoring of such features is a difficult process itself.

This raises the question whether the observed problems are inherent to feature-refactoring, or whether they are caused by AspectJ. For example, the needed statement extensions, the necessity to access local variables, and the strong dependence on implementation details might be reasonably dismissed as inherent complexity of refactoring features.

The important question is: How does AspectJ support the already complex task? Our observation is that AspectJ adds complexity, which is not inherent in feature-refactoring. Fragility, hard to read and maintain specifications, and strong-dependence on tool support are contributing factors.

AspectJ may still be useful for implementing features in an SPL that was designed to be extensible, where extension points were anticipated or naming conventions could be used to exploit the strengths of AspectJ. This is a topic of future study.

6. Related Work

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

An alternative is the use of *collaborations* with *mixins* to implement features [9]. Other concepts include *subjects* [25], *virtual classes* [38, 20], *nested inheritance* [42] and *classboxes* [10].

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 feature modules* [5, 2], *aspectual collaborations* [33], and *aspect components* [7].

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

Czarnecki and Eisenecker first suggested to use aspects to implement features [17]. An early study on decomposing features with aspect-oriented languages including AspectJ was performed by Murphy et al. [40]. This study focuses on expressibility of language constructs and notes 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. noticed problems in using aspects to implement features because of a lack of infrastructure and means to describe coherent feature implementations containing multiple aspects [36].

Colyer et al. discussed how features could be separated with AspectJ to create program families, on a theoretical level [16]. A brief discussion about the quality of the resulting features with AspectJ was already initiated by Nyssen et al., based on the obliviousness and the implicit extensions of aspects [41]. Although a 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 some empirical case studies on refactoring existing applications, mainly from the areas of embedded database systems and middleware systems.

The work closest to ours is the aspect-oriented refactoring of the embedded database engine *HSQLDB* into nine features, done as case study by Eibauer [19]. Even though the feature selection was based on a catalog of typical crosscutting concerns that are supposed to be encapsulated by AspectJ easily, the results are similar: the number of homogeneous crosscuts was lower than expected, code quality decreased for heterogeneous extensions, and many limitations we observed were found as well.

Tesanovic et al. refactored the C version of *Berkeley DB* into four features with an AspectJ-like language [46]. This study showed the general possibility and the advantages of having a configurable version of Berkeley DB from a database perspective, but did not focus on the feature implementations and their quality.

Zhang and Jacobsen discussed the refactoring of five selected features of a middleware platform into aspects [47]. They call this process *horizontal decomposition*, and their aim is to configure the platform by including or excluding features. However, they focus only on some selected crosscutting concerns found with aspect mining tools.

Similarly, Hunleth and Cytron extracted 21 features from a small middleware platform to make it configurable [26]. They used their own infrastructure with distinct feature directories built on *ant*, which is similar to our solution. They focused on the differences in footprint and performance compared to optional feature implemented with object-oriented design patterns and observed improvements.

Coady and Kiczales refactored four small crosscutting concerns in *FreeBSD* into features, three of which are optional and can be used to configure *FreeBSD* [14], examining the evolution of the aspects.

All these case studies are smaller than our refactorings of Berkeley DB: Even though some case studies (e.g., *HSQLDB*, *FreeBSD*) 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 their feature models are very simple. No case study analyzed the effects of scale. Furthermore, most studies focused on refactoring (homogeneous) crosscutting concerns, thus 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 were usually evaluated only based on code size or simple coupling metrics. Readability, fragility, or necessity of tool support were not considered.

7. Conclusions

A common way in which SPLs are created is via an extractive approach, where one (or more) applications serve as a baseline. By factoring optional features, variations on an application can be created by feature removal. This is the approach that we took to refactoring Berkeley DB. As a platform, we used AspectJ to implement our features.

Feature-refactoring legacy applications is a *very* difficult problem, because such applications—even if they had an elegant design to begin with—do not imply that their design was amenable to feature-extensibility. It places great stress on *any* language or tool. Nevertheless, our experience with Berkeley DB suggests that AspectJ is not an appropriate language for this task.

We used only the basic capabilities of AspectJ, mainly static introductions and method extensions. Advanced language constructs which are unique to AspectJ, like conditional extensions or homogeneous extensions, were rarely applicable. Even so, readability and maintainability of the resulting code was clearly a problem. Pointcut fragility, a well-known issue, was very evident, and ensuring correct weaving was hard. Tool support that might be able to reduce these problems is not yet sufficiently available.

As other researchers use AspectJ in larger applications, we believe they will encounter problems that are similar to those that we identified in this paper. We hope that our work, and subsequent confirmations of our observations, can serve as a starting point to improve languages and tool support that developers need for creating SPLs by feature-refactoring legacy applications.

Acknowledgments. We thank Roberto Lopez-Herrejon and Maximilian Störzer for their helpful comments on earlier drafts of the paper. This paper is a condensation of the master's thesis of C. Kästner. Apel was sponsored by the German Research Foundation (DFG), project number SA 465/31-1 and SA 465/32-1. Batory's research is sponsored by NSF's Science of Design Project #CCF-0438786.

References

[1] J. Aldrich. Open Modules: Modular Reasoning About Advice. In *Proc. Europ. Conf. Object-Oriented Programming*. 2005.

[2] S. Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, University of Magdeburg, 2007.

[3] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proc. Int'l Conf. Generative Programming and Component Engineering*, 2006.

[4] S. Apel, C. Kästner, T. Leich, and G. Saake. Aspect Refinement - Unifying AOP and Stepwise Refinement. In *Proc. Int'l Conf. TOOLS EUROPE*, 2007.

[5] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proc. Int'l Conf. on Software Engineering*. 2006.

[6] S. Apel and J. Liu. On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In *ECOOP Workshop on Aspects, Dependencies, and Interactions*, 2006.

[7] I. Aracic et al. Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, 3880, 2006.

[8] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conference*, 2005.

[9] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6), 2004.

[10] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: controlling the scope of change in Java. In *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications*. 2005.

[11] D. Binkley et al. Tool-Supported Refactoring of Existing Object-Oriented Code into Aspects. *IEEE Transactions on Software Engineering*, 32(9), 2006.

[12] J. R. Carter. Ada's design goals and object-oriented programming. *Ada Lett.*, XIV(6), 1994.

[13] P. Clements and C. Kreuger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4), 2002.

[14] Y. Coady and G. Kiczales. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proc. Int'l Conf. Aspect-Oriented Software Development*. 2003.

[15] L. Cole and P. Borba. Deriving Refactorings for AspectJ. In *Proc. Int'l Conf. Aspect-Oriented Software Development*, 2005.

[16] A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical Report COMP-001-2004, Computing Department, Lancaster University, 2004.

[17] K. Czarnecki and U. W. Eisenecker. Synthesizing Objects. In *Proc. Europ. Conf. Object-Oriented Programming*. 1999.

[18] M. Eaddy and A. Aho. Statement Annotations for Fine-Grained Advising. In *ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, 2006.

[19] U. Eibauer. Studying the Effects of Aspect Oriented Refactoring on Software Quality using HSQLDB as Example. Master's thesis, University of Passau, Germany, 2006.

[20] E. Ernst, K. Ostermann, and W. Cook. A virtual class calculus. In *Proc. Conf. Principles of Programming Languages*. 2006.

[21] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[22] M. L. Griss. Implementing Product-Line Features by Composing Aspects. In *Proc. Int'l Software Product Line Conference*. 2000.

[23] K. Gybels and J. Brichau. Arranging Language Features for more Robust Pattern-Based Crosscuts. In *Proc. Int'l Conf. Aspect-Oriented Software Development*. 2003.

[24] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of Aspect-Oriented Software. In *Proc. Net.ObjectDays*, 2003.

[25] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. *SIGPLAN Not.*, 28(10), 1993.

[26] F. Hunleth and R. K. Cytron. Footprint and feature management using aspect-oriented programming techniques. In *Proc. Conf. Languages, Compilers and Tools For Embedded Systems*. 2002.

[27] K. Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA, USA, 1990.

[28] C. Kästner. Aspect-Oriented Refactoring of Berkeley DB. Master's thesis, University of Magdeburg, Germany, 2007.

[29] C. Kästner, S. Apel, and G. Saake. Implementing Bounded Aspect Quantification in AspectJ. In *ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, 2006.

[30] G. Kiczales et al. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*. Springer, 1997.

[31] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[32] K. Lee et al. Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development. In *Proc. Int'l Software Product Line Conference*. 2006.

[33] K. J. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *Comput. J.*, 46(5), 2003.

[34] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proc. Int'l Conf. on Software Engineering*, 2006.

[35] R. Lopez-Herrejon and S. Apel. Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering*, 2007.

[36] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. Europ. Conf. Object-Oriented Programming*. 2005.

[37] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*. 2006.

[38] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A powerful mechanism in object-oriented programming. In *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications*. 1989.

[39] M. P. Monteiro and J. M. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proc. Int'l Conf. Aspect-Oriented Software Development*, 2005.

[40] G. C. Murphy et al. Separating Features in Source Code: an Exploratory Study. In *Proc. Int'l Conf. on Software Engineering*. 2001.

[41] A. Nyssen, S. Tyszberowicz, and T. 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*, 2005.

[42] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *Proc. Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages and Applications*. 2004.

[43] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming*. 1997.

[44] T. Reenskaug et al. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *J. OO Programming*, 5(6), 1992.

[45] M. Störzer and J. Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *Proc. Int'l Conf. Software Maintenance*. 2005.

[46] A. Tesanovic, K. Sheng, and J. Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *Proc. Int'l Database Engineering and Applications Symposium*. 2004.

[47] C. Zhang and H.-A. Jacobsen. Quantifying Aspects in Middleware Platforms. In *Proc. Int'l Conf. Aspect-Oriented Software Development*. 2003.