Granularity in Software Product Lines

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

Sven Apel
Department of Informatics and Mathematics
University of Passau, Germany
apel@uni-passau.de

Martin Kuhlemann
School of Computer Science
University of Magdeburg, Germany
mkuhlema@ovgu.de

ABSTRACT

Building *software product lines (SPLs)* with features is a challenging task. Many SPL implementations support features with coarse granularity – e.g., the ability to add and wrap entire methods. However, fine-grained extensions, like adding a statement in the middle of a method, either require intricate workarounds or obfuscate the base code with annotations. Though many SPLs can and have been implemented with the coarse granularity of existing approaches, fine-grained extensions are essential when extracting features from legacy applications. Furthermore, also some existing SPLs could benefit from fine-grained extensions to reduce code replication or improve readability. In this paper, we analyze the effects of feature granularity in SPLs and present a tool, called *Colored IDE (CIDE)*, that allows features to implement coarse-grained and fine-grained extensions in a concise way. In two case studies, we show how CIDE simplifies SPL development compared to traditional approaches.

Categories and Subject Descriptors: D.2.3 [**Software**]: Software Engineering—*Coding Tools and Techniques*; D.2.6 [**Software**]: Software Engineering—*Programming Environments*; D.3.3 [**Software**]: Software Engineering—*Language Constructs and Features*

General Terms: Design, Languages

Keywords: Software product lines, virtual separation of concerns, feature refactoring, IDE

1. INTRODUCTION

A *software product line (SPL)* aims at generating tailored programs from a set of features. Each feature represents an increment in functionality relevant to stakeholders. Different programs tailored for a given task or environment can be created by selecting a particular subset of features. Using SPLs it is possible to create program families of related programs for a domain.

A feature's implementation extends the program in one or more places. In prior work, we successfully created SPLs, e.g., [1, 22, 30], with several different implementation approaches like AHEAD [4], mixin layers [44], aspectual feature modules [2], or aspects [24]. At that time, we have not been aware of the importance of granularity on feature implementation.

In a recent case study, we decomposed a legacy application – *Berkeley DB*, an embedded database engine[1] – into features in order to transform it into an SPL [22]. We factored out 38 features, so that users could configure Berkeley DB to optionally include features such as *transactions*, *statistics*, or *logging*. We noticed that this decomposition was much more difficult than our prior SPL implementations. Several features were hard to implement because they needed to introduce statements in the middle of methods, or add parameters to existing methods. The modularization of features became very difficult to understand and maintain. Though some problems were caused by limitations of AspectJ, we found that other languages like Jak [4], Jiazzi [36], or Hyper/J [49] also poorly support extensions of such fine granularity.

What we observed is that languages that are based on *collaboration-based design* [40] focus on adding new members to existing classes, new classes, and extending existing methods [3]. These capabilities are sufficient for many applications. But for decomposing Berkeley DB, we needed much more fine-grained extensions. For example, when adding transaction and synchronization mechanisms to the database system, we extended the program in over 240 places, often in places not trivially to access, like in the middle of a method [22]. To realize such extensions, we had to use workarounds that obfuscated the code. In the end, we even considered restarting from scratch using preprocessor directives such as *'#ifdef'* and *'#endif'* to surround feature-specific code, but refrained because of other problems this would have introduced [45], which we discuss later.

Fine-grained extensions pose a major challenge for current approaches to feature-based SPL development, especially when creating a SPL by decomposing a legacy application. Alerted by these results we analyzed the granularity of extensions in SPLs we created earlier. Even though fine-grained extensions were required less frequently than in Berkeley DB, we found potential for improvements where current implementations replicate code or use workarounds that obfuscate the source code. We did not notice this potential earlier because we accepted the limitations of the given languages.

In this paper, we explore effects of granularity of different approaches on SPL development. We show that existing approaches are not able to implement fine-grained extensions satisfactorily and analyze possible solutions that allow implementing SPLs without sacrificing understandability. We present a tool, called *Colored IDE*

ICSE'08, May 10–18, 2008, Leipzig, Germany.

[1] http://www.oracle.com/database/berkeley-db

(CIDE), that combines the strengths of existing approaches and adds support to overcome the granularity problem. Finally, we illustrate the benefits of CIDE in two case studies, the initial Berkeley DB case study and a small, existing SPL, in which we did not realize granularity effects at first.

2. BACKGROUND: APPROACHES TO SPL IMPLEMENTATION

There are two common ways to implement an SPL: the *compositional approach* and the *annotative approach*.

2.1 The Compositional Approach

Compositional approaches implement features as distinct modules. To generate a product line member, a set of modules is *composed*, usually at compile-time or deploy-time. There is a large body of work on feature composition usually employing component technologies [48], or specialized architectures and languages like frameworks [20], mixin layers [44], AHEAD [4], multi-dimensional separation of concerns [49], and aspects [24].

In Figure 1, we show an example of 3 modules implemented with Jak (AHEAD Tool Suite) [4], a compositional approach, that uses mixin technologies. The first module (Lines 1–5) implements the basic stack. The other two modules implement extensions for the two features locking and logging. In Jak a feature can extend an existing class using the *refines* keyword and introduce new methods or extend existing ones with wrappers (by overriding using the *Super* keyword to call the original implementation).

```
1  class Stack {
2    boolean push(Object o) {
3      elementData[size++] = o;
4    }
5  }
```

```
6  refines class Stack {
7    boolean push(Object o) {
8      Lock l=lock(o);
9      Super.push(o);
10     l.unlock();
11   }
12   Lock lock(Object o) { /*...*/ }
13 }
```

```
14 refines class Stack {
15   boolean push(Object o) {
16     Super.push(o);
17     log("added " + o);
18   }
19   void log(String msg) { /*...*/ }
20 }
```

Figure 1: A basic stack and two features implemented in Jak.

2.2 The Annotative Approach

Annotative approaches implement features with some form of explicit or implicit annotations of the source code. Typical examples of explicit annotations are *'#ifdef'* and *'#endif'* statements of C/C++ style preprocessors to surround feature code. Others use Java annotations or new language constructs in the code to be extended. Examples are *explict programming* [7], *Frames/XVCL* [18], *Spoon* [39], *Gears* [28], *software plans* [9], *metaprogramming with traits* [52], and *aspects using annotations* [27]. Alternatively, implicit annotations exploit existing language facilities. For example, deliberately introduced empty methods can be used as hooks for extensions, or naming conventions can be employed for extensions like "synchronize all methods starting with *'sync_'*". In some approaches, features are partly modularized leaving just annotations in the code, at which feature code is later introduced, while in others the whole SPL including alternative and mutually exclusive features are encoded in a single code base and configurations are created by removing or deactivating code fragments.

Especially common is the use of preprocessors to configure a program. For example, with the C implementation of Berkeley DB a user can configure 11 different features at compile-time. An example code fragment from Berkeley DB with *'#ifdef'* statements for the features *HAVE_QUEUE* and *DIAGNOSTIC* is shown in Figure 2.

```
1  static int __rep_queue_filedone(dbenv, rep, rfp)
2    DB_ENV *dbenv;
3    REP *rep;
4    __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6    COMPQUIET(rep, NULL);
7    COMPQUIET(rfp, NULL);
8    return (__db_no_queue_am(dbenv));
9  #else
10   db_pgno_t first, last;
11   u_int32_t flags;
12   int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14   DB_MSGBUF mb;
15 #endif
16   // over 100 lines of additional code
17 #endif
18 }
```

Figure 2: Code excerpt of Berkeley DB.

Most SPL implementations can be categorized into compositional and annotative approaches. Additionally, some approaches like aspects can be included in both groups, depending on how they are used, or combine approaches from both groups. However, there are techniques that do not fit into either group, e.g., those based solely on tool support [6], based on *model-driven development* [51], or on *generative programming* [10]. Still, in this paper we focus on compositional and annotative approaches because they cover a significant fraction of contemporary SPL implementations and reveal interesting differences concerning granularity of extensions.

3. GRANULARITY

Features can extend a program with additional code. Extensions with coarse granularity add new classes or methods to the program or extend explicit extension points. Plug-in architectures in frameworks and component approaches are typical approaches that provide possibilities for such extensions. However, this might not be sufficient. Developers might want to introduce new statements into existing methods and extend expressions or even method signatures. These are fine-grained extensions and require additional support. For example, they arise when legacy applications are feature-refactored [32]. (These applications were not designed with feature granularity in mind, and consequently fine-grained extensions are commonly needed when extracting features). We evaluate possible levels of granularity and typical problems for both compositional and annotative approaches.

3.1 Compositional Approaches

Existing compositional techniques usually allow coarse-grained extensions only. Many provide mechanisms to define explicit extension points, which are the only points that can be extended by

a feature. Others like AHEAD and AspectJ can extend virtually every method of the system. In most approaches it is only possible to introduce new classes, methods, or fields and to extend whole methods using wrappers (a.k.a. around advice, method refinements, or method overriding) [3].

```
1 class Stack {
2   void push(Object o, Transaction txn) {
3     if (o==null || txn==null) return;
4     Lock l=txn.lock(o);
5     elementData[size++] = o;
6     l.unlock();
7     fireStackChanged();
8   }
9 }
```

Figure 3: Fine-grained extension example.

We [22] and others [37] noticed several limitations when implementing fine-grained extensions compositionally. We exemplify the three most common limitations by means of the code snippet in Figure 3, in which the underlined code belongs to a synchronization feature and should be implemented in its own module.

1. **Statement Extensions.** In most compositional approaches it is not possible to introduce statements in the middle of an existing method in order to extend certain statements or sequences of statements therein.[2] For example, consider how to synchronize only the statement in Line 5. Simple wrappers around the whole method are not sufficient. Instead, we have to introduce the locking statements in Lines 4 and 6 specifically. Note, statement extensions might also access local variables. Usually, workarounds borrow from annotative approaches and introduce artificial extensions points for extension. Typically, a developer would introduce calls to empty hook methods [37] or perform an Extract Method refactoring [13] that moves Line 5 to its own method so that it can be extended with a wrapper. Local variables, if accessed in the extension, are passed as parameters. Either workaround requires explicit or implicit annotations and severely obfuscates the source code [37, 22].
2. **Expression Extensions.** Extensions to an individual expression can occur as well. An example is shown in Line 3, in which the condition of the *if* statement is extended. A typical workaround again creates a new method and moves the expression there, so that it can be extended with wrappers.
3. **Signature Changes.** To the best of our knowledge, there is no compositional approach that allows to introduce an additional parameter into an existing method signature, as the *txn* parameter in Line 2. Instead, method signatures are considered unchangeable. Typical workarounds store the additional parameters in thread-safe fields, duplicate code, or use complex language mechanisms like the Wormhole Pattern in AspectJ [29]. However, all of these workarounds introduce different problems and reduce code quality [42, 21]. Note that it is also necessary to adapt all calls to the extended method.

A possible implementation of the base code and its extensions using Jak is shown in Figure 4. Statements and expression extensions are implemented with the two hook methods *h1* and *h2*. The parameter is passed with a thread-safe field *pushTxn* and the original *push* method is deactivated by throwing an exception. Apparently, the extension code in Figure 4 is much larger than the amount of underlined code in Figure 3.

[2] A notable exception is *AspectJ* that enables to extend method calls or field access inside specific methods [25, 22]. This feature can be used to emulate statement extensions in some cases (cf. Sec. 5.1).

```
1  class Stack {
2    void push(Object o) {
3      if (o==null || h1()) return;
4      h2(o);
5      fireStackChanged();
6    }
7    boolean h1() { return false; }
8    void h2(Object o) {
9      elementData[size++] = o;
10   }
11 }
```

```
12 refines class Stack {
13   ThreadLocal<Transaction> pushTxn =
         new ThreadLocal<Transaction>();
14   void push(Object o, Transaction txn) {
15     pushTxn.set(txn);
16     Super.push(o);
17   }
18   void push(Object o) {
19     throw new UnsupportedOperationException(
20         "Call push(Object,Transaction) instead");
21   }
22   boolean h1() {
23     return pushTxn.get() == null;
24   }
25   void h2(Object o) {
26     Lock l = pushTxn.get().lock(o);
27     Super.h2(o);
28     l.unlock();
29   }
30 }
```

Figure 4: Fine-grained extension with AHEAD.

Extending compositional approaches with new language constructs for fine-grained extensions is not trivial either because of several conceptual problems. Firstly, signatures are used to identify the methods that are to be extended. If changing method signatures for an optional feature was possible, another naming scheme would need to be used to identify methods. Consequently, most languages consider signatures as immutable, and do not account for the possibility of signature changes.

Secondly, compositional approaches only introduce new code fragments in positions in which the order does not matter. Thus, it is possible to introduce new classes into the program or new methods into a class, but not new statements at a fixed position inside a method. This target position is not known when implementing the feature and could move if other features introduced statements as well. Therefore, compositional approaches usually offer only wrappers that add statements at the beginning and/or the end of a method, but not at a finer granularity. Similarly, parameters in method signatures are ordered, which makes parameter introductions difficult.

The coarse granularity of compositional approaches leads to several problems when developers use them to implement fine-grained extensions anyway. Workarounds often obfuscate the source code and are verbose and hard to understand. Many workarounds replicate code or use heavy-weight architectures that induce performance penalties [37, 22, 42, 21].

3.2 Annotative Approaches

Conceptually, annotations can mark code fragments at arbitrary levels of granularity. They simply introduce markers at the exact positions that should be extended. Typical examples are C/C++ style preprocessors, which although they annotate only whole physical lines, are sufficient for even the finest extensions due to the ability

to isolate language constructs in separate lines. A possible preprocessor implementation of our example from Figure 3 is shown in Figure 5. Other annotation approaches allow similarly fine-grained extensions, e.g., [19, 9].

```
1  class Stack {
2    void push(Object o
3  #ifdef TXN
4    , Transaction txn
5  #endif
6    ) {
7      if (o==null
8  #ifdef TXN
9        || txn==null
10 #endif
11       ) return;
12 #ifdef TXN
13     Lock l=txn.lock(o);
14 #endif
15     elementData[size++] = o;
16 #ifdef TXN
17     l.unlock();
18 #endif
19     fireStackChanged();
20   }}
```

Figure 5: Fine-grained extension with C/C++ preprocessor.

Annotations do not share the conceptual limitations regarding ordered statements and fixed signatures because they indicate the final position in the base code. Therefore, a method can always be identified by its final signature and also the position of a statement or a parameter in an ordered list can always be determined.

Still, our experience[3] and reports by others, e.g., [45, 5], show that annotations have problems as well. Firstly, annotations themselves obfuscate the source code as apparent in Figure 5. Secondly, annotating arbitrary code fragments, whether they make sense or not, is problematic. For example, it is possible to annotate an opening bracket with one feature and the closing bracket with another. This makes annotations error-prone and raises complexity. Thirdly, there may be problems dealing with separating terminals like commas between parameters. There are frequent situations when such simple syntactic elements must be annotated for features as well.

In Figure 6 we show an example: an *init* method with two parameters, in which the first parameter is included only if *transactions* are enabled in the system, and the second is included only if *logging* is enabled. However, when annotating this code fragment with C/C++ style preprocessors we have to split the method declaration into multiple lines and even include the comma inside a nested *'#ifdef'* statement so that all derivable variants are syntactically correct.

Despite these problems, annotative approaches support fine-grained extensions better than compositional approaches. However, they provide no perceptible form of modularity.

4. CIDE

4.1 Overview

Motivated by the problems of both compositional and annotative approaches, we built an Eclipse-based prototype tool for decomposing legacy applications into features that may have a fine granularity.[4] It uses the semantics of preprocessors, i.e., it can be classified as

[3]In the FAME-DBMS project, colleagues analyzed and decomposed the C version of Berkeley DB, which employs an annotation approach (funded by the German Research Foundation, project no. SA 465/32-1).

[4]The tool can be downloaded at `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide`.

```
1  void init(Transaction txn, LoggingLevel level){/*impl.*/}
```

```
2  void init(
3  #ifdef TRANSACTION
4    Transaction txn
5  #ifdef LOGGING
6    ,
7  #endif
8  #endif
9  #ifdef LOGGING
10   LoggingLevel level
11 #endif
12   ){/*impl.*/}
```

Figure 6: Decomposition with preprocessors.

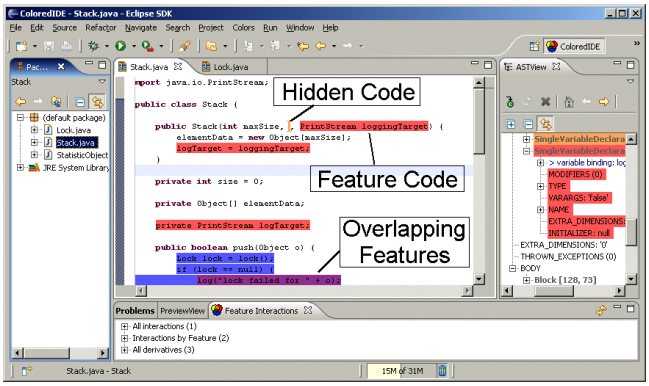

Figure 7: CIDE Screenshot.

annotative approach, but avoids the pollution of source code. Developers start with a fully composed application with all features implemented in a single code base, typically a legacy application. Then, they make features explicit by successively associating code fragments with one or more features, i.e., they mark the corresponding code. Alternatively, developers can also extend the application with new features and associate all new code fragments with those features. Just as with *'#ifdef'* statements, code fragments are only included when all associated features are selected in a given configuration.

In contrast to traditional preprocessors as in C/C++, we do not obfuscate the source code with additional annotations. Instead, we use the representation layer of the editor to indicate the associated features with different background colors. Thus, developers can directly recognize whether a code fragment is associated with a feature. In case a code fragment is associated with multiple features, which is traditionally done with nested preprocessor statements, we mix the according background colors (e.g., red + blue = purple). Feature names are shown in tool-tips on request. Note that it is usually not possible to recognize the features of a code fragment solely by background colors, especially when many features overlap. However, colors are sufficient to determine the beginning and the end of a code fragment associated with a set of features, and it is convenient to look up the actual features using tool-tips. Because of its colorful appearance (cf. Fig. 7) we named the tool *Colored Integrated Development Environment (CIDE)*.

As with preprocessors it is still possible to insert or edit code, while the colors remain assigned to the code fragments. But even though CIDE is based on preprocessor semantics, we do not assign features to arbitrary code fragments to avoid the problems of meaningless associations and syntactical elements illustrated in Sec-

tion 3.2. Instead, we assign features to *structural code elements* of the source code. The form of such structural elements depends on the artifact type. For example, programs in many languages can be expressed as *abstract syntax tree (AST)*, not only Java programs. In all these cases we use this underlying structure and associate features to structural elements.

4.2 Coping with Feature Granularity

We illustrate the mechanisms used in CIDE by the example of Java code and its AST representation. An AST contains all structural elements that are relevant for the source code. It is possible to recreate the source code from the AST (except its original formatting). In Figure 8, we exemplify a small Java code snippet and a corresponding AST. Note, the AST does not include syntactical elements like commas or brackets.

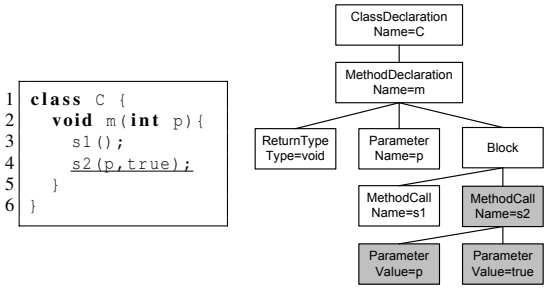

Figure 8: AST Example.

Instead of using offset and length to specify code fragments, we assign features directly to subtrees of the AST as defined by the language grammar. For example, we can assign a feature to a class node and all its children (i.e., making a class optional), or to a statement and its children as highlighted in Figure 8 (i.e., statements are optional child nodes of *block* nodes). It is even possible to assign a single parameter node to a feature (i.e., an optional child node of a method node). However, nodes that are not optional in the AST (i.e., that cannot be removed without invalidating the AST), like the return type of a method cannot be associated with features individually. Developers can but are not required to view the code as AST, usually, in the editor they assign the selected code fragments directly to features, which CIDE automatically maps to AST elements.

For Java code, we carefully defined two kinds of exceptions from the subtree rule exemplified in Figure 9. Firstly, statements that embrace other statements can be associated with features without necessarily associating its child elements. So, it is possible to remove individual *if*, *for*, *while*, *try*, and similar statements without removing the statements they surround. Secondly, children of binary expression nodes can be associated with features even though they are not optional AST elements (every binary expression node requires two child expressions). This allows us do decompose expression statements as in Figure 3. We decided to include these exceptions for increased granularity.

The restriction of assigning only optional AST subtrees to features enforces decomposition into 'reasonable' code fragments. Thus, developers can no longer associate only an opening bracket but not the closing one with a feature. Furthermore, they do not need to deal with syntactical elements such as commas in Figure 6 at all. We allow only AST operations that create a well-formed AST as output, so the Java compiler can *parse* the generated code of any configuration. At the same time, the AST structure is extremely

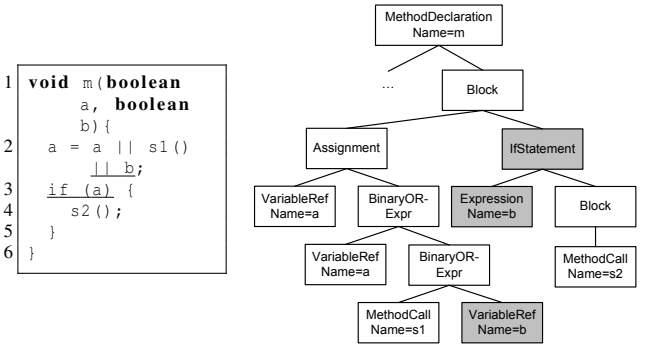

Figure 9: Exceptions to the optional subtree rule.

flexible, allowing developers to associate arbitrary statements or even parameters or parts of expressions with features. Decomposing a code snippet like the one introduced in Figure 3 is straightforward, because parameters, statements and expressions are all AST nodes that can simply be 'colored'.

Note, the granularity of CIDE is not as fine as that provided by C/C++ like preprocessors. For example, developers cannot specify two alternative result types of a method, because the AST only allows one child element for the result type per method. In practice we have not yet found any cases where this was a limitation. Furthermore, for reasons of simplicity developers can assign only a set of features to a code fragment and not a propositional formula [11]. Consequently, it is not possible to specify a code fragment to be included if a feature is *not* selected (*#ifndef*). However, we are able to work around these problems by modeling features differently in a feature model (e.g., specify two alternative features instead of a single optional one). If future case studies showed the demand for such expressibility, we could still extend CIDE at the cost of a more complicated user interface.

4.3 Feature Management

Feature implementations with fine-grained extensions are difficult to understand in both compositional and annotative techniques, as illustrated in Section 3. Even though compositional approaches can implement features cohesively, workarounds make it hard to understand extensions. Also the feature code in annotative approaches is hard to understand because it is scattered and annotations obfuscate the source code.

CIDE enhances feature management (navigation, selection, composition) with tool support. Firstly, navigation support allows developers to jump between code fragments associated with a certain feature set to diminish the problems of missing feature cohesion. This makes it easy to find all code fragments associated with a feature. Because feature code is still placed where it extends the program, and it is therefore obvious to see how it extends the program, it is simple to understand how a feature is implemented (cf. Fig. 3). The navigation support is in line with prior approaches of *concern graphs* [41] and the *map metaphor* [14] that choose tool support over complex modular implementations.

Secondly, a projection facility can hide all code associated with a feature in the editor during development, so that the remaining code can be viewed in isolation. The projection facility is implemented like code folding in modern development environments, in which bodies of methods or comments can be folded and unfolded on request. When the developer requests a feature to be hidden, CIDE just leaves a marker to indicate hidden code. Thus, the developer can focus on selected features and hide feature code that is not relevant

to the current task. The markers are still useful for modifications as they alert the developer of feature extensions that might be necessary to adapt as well.

Finally, we implemented a mechanism to export the marked code into cohesive modules implemented with compositional approaches [23]. Our prototype currently provides exports to Jak and to AspectJ. This way, feature implementations can be used independently of CIDE, e.g., for further code generation and optimization steps.

We argue that, compared to existing compositional and annotative approaches, CIDE makes it easier to develop feature-based SPLs because it hides all programming language concepts of refinements, aspects, hook methods, annotations, and similar for implementing features that are necessary in these approaches. It provides different views on the source code. Developers can hide features, navigate between them, and even export them. Even though CIDE cannot replace cohesive modules, it supports understanding scattered feature implementations and enables developers to choose the best view on the source code for each task.

5. CASE STUDIES

In order to support our proposal, we conducted two case studies. Firstly, we analyze our decomposition of the embedded database engine Berkeley DB, a large scale legacy application with 84,000 LOC. Secondly, we study a small SPL of graph algorithms with 2,000 LOC designed as an SPL using a compositional approach. In both, we compare their implementations with CIDE. The code for both case studies is available for download from the CIDE web page.[5]

5.1 Berkeley DB

Our initial insight that the granularity of existing compositional approaches was insufficient came from a case study, in which we decomposed a large scale legacy application - the embedded database engine Berkeley DB JE – into 38 features to make features explicit in its design and to make it configurable as an SPL. To decompose Berkeley DB, we first located and marked feature code, then removed this feature code, and finally reintroduced it using feature modules implemented with AspectJ [25], a compositional approach. Due to space restrictions, we limit our report to statistics and the most important insights in anecdotical form.

Berkeley DB was not designed as an SPL with features in mind. Instead it was implemented as a single application with a clean object-oriented design modularized in classes and packages. Some configurability was achieved using external parameters and runtime checks. Consequently, feature code was scattered all over the application. Some larger features like Synchronization and Transaction affected up to 30 (of 300) classes in Berkeley DB in over 150 places (cf. [21, 22] for detailed statistics).

Decomposition with AspectJ. When decomposing Berkeley DB we were confronted with fine-grained extensions in almost every feature. For example, feature code was often located right in the middle of a method. Of 1,144 extensions used to implement the 38 features, 640 extensions (56 %) introduced new classes, methods, or fields. 214 other extensions (19 %) were simple method extensions that added wrappers to existing methods. They were well supported by AspectJ. However, 261 extensions (23 %) were required at statement level and 24 (2 %) at expression level, which posed major problems.

Statement extensions were implemented in two different ways. Firstly, AspectJ supports extensions finer than just wrappers around methods: it is possible to extend methods *calls* or field accesses inside specific methods by combining *call*, *set*, or *get* with *withincode* pointcut designators. We used such extensions to emulate statement extensions when the feature code was placed directly before, after, or around a single method call or field access. This was sufficient for 121 extensions (46 % of all statement extensions). However, our solution was rather fragile, because any changes to the internal implementation of the base code might require the pointcuts to be altered. For the remaining 140 statement and 24 expression extensions we had to introduce hook methods and extend those.

We also faced the problem that certain method parameters of the code base belonged to a feature and should be removed when the feature is not selected. This made it very difficult to detach some features in Berkeley DB, because AspectJ, like other compositional approaches, does not permit changes to method signatures. We first noticed the problem when we decomposed the part of the transaction system that is responsible for atomicity. To group several operations as a single transaction, the user can create a *Transaction* object and pass it to operations like *put*, *get*, or *delete*. This object is then passed further along the control flow to acquire suitable locks. Inside Berkeley DB, such a parameter is included in the signature of 59 methods. Without the transaction feature, none of these methods should include a transaction parameter.

We experimented with different possibilities to implement the Transaction feature. We removed the parameter from the base code and introduced a second method for each operation with the additional parameter that calls the first method. We could pass the parameter in a thread-protected field, use method objects [13], use dummy or default parameters, employ AspectJ's wormhole pattern [29], or use parameter objects that encapsulate all parameters. Unfortunately, all options obfuscate the source code. Firstly, excited by AspectJ's advanced constructs, we implemented the wormhole pattern in 16 pieces of advice as exemplified in Figure 10. There, all calls to *getWritebleLocker(Environment)* are replaced by calls to *getWriteableLocker(Environment, Transaction)*, and the transaction parameter is captured from interface methods like *openDatabase* and *get* using the *cflow* pointcut designator. Eventually, we found this too fragile and too hard to understand [22]. Finally, we settled with parameter objects named *OperationContext* and interface methods that created those objects for internal use. To extend the method, we only needed to extend the parameter objects.

While we could work around the parameter limitation for the transaction feature, we did not attempt any further decomposition of features, in which a large amount of parameters were involved. Especially the Locking feature appeared unmanageable with compositional approaches as it had to introduce parameters like *locker*, *lockMode*, or *lockType* 289 times. Decomposing the Locking feature would either result in utterly unreadable code or require a complete preliminary redesign of the whole database engine.

Together, workarounds for fine-grained extensions of statements, expressions, and parameter lists made the decomposition very difficult and the implementation tedious. Though it was possible to decompose most features, code quality suffered and the resulting code base became hard to understand and maintain.

Decomposition with CIDE. After these discouraging initial results, we experimented with other compositional approaches and with decomposing Berkeley DB using a C/C++ preprocessor but limitations discussed in Section 3.2 held us from trying them on a case study of this size. Finally, we decomposed Berkeley DB once more into the same features using CIDE. In CIDE, fine-grained extensions were no problem, because we could directly associate single statements, exceptions, or even parameters with features. This avoids all previously required workarounds. Because we did not need to

[5] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide

```
1 pointcut interfaceCallWithTransaction(Transaction txn) :
2     (execution(* Environment.openDatabase(..)) && args(txn,*,*)) ||
3     (execution(* Database.get(..)) && args(txn,*,*,*)) || ...;
4 pointcut getWritableLocker(Environment env): call(Locker LockerFactory.getWritableLocker(Environment)) && args(env);
5
6 Locker around(Environment env, Database db, Transaction txn) throws DatabaseException :
7     getWritableLocker(env) && this(db) && cflow(interfaceCallWithTransaction(txn))
8     { return LockerFactory.getWritableLocker(env, txn); }
```

Figure 10: Using the Wormhole Pattern to pass a parameter.

perform any manual decomposition or implementation of feature modules, but only assign features to code fragments inside the IDE, the decomposition of Berkeley DB was more convenient and much faster (3 days instead of 1 month)[6]. CIDE's finer granularity supported decomposing Berkeley DB significantly. We were even able to decompose the Locking feature with its 289 parameter extensions.

By using CIDE, understanding the SPL became simpler, because neither do additional statements obfuscate the source code, nor is it necessary to understand complex workarounds like in Figures 4 and 10. Moreover, the projection facilities helped to reason about features in isolation or in concert with other features.

5.2 The Graph Product Line

The *graph product line (GPL)* is a small SPL of graph algorithms with about 2,000 LOC. It was designed from scratch as an SPL and suggested as a benchmark for SPL technologies [33]. The domain of graphs was chosen because it is well understood and the algorithms are well known in computer science. It consists of 14 features: edges can be directed, undirected and optionally weighted, there are two search methods breadth first search (BFS) and depth first search (DFS), and finally there are several graph algorithms like cycle checking, shortest path, strongly connected, or minimum spanning tree. The implementation as an SPL makes it possible to select the graph properties and algorithms needed for a given problem [33]. The original version of GPL was implemented with mixin layers [44], but other implementations e.g. in AspectJ [34] and Hyper/J [35] are available. All these implementations use compositional approaches with similar results regarding the following analysis.

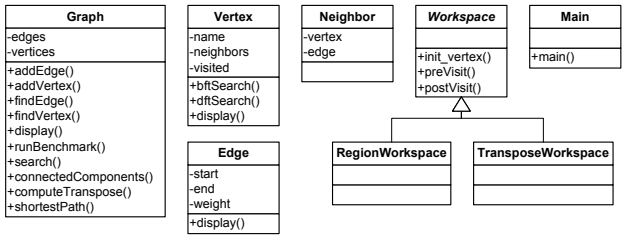

Figure 11: UML class model of GPL (excerpt).

Although the GPL case study and the figures below appear almost trivial, we intentionally chose it because it is well suited to show that there are benefits from implementation approaches that support fine-grained extensions, *even* for SPLs that are (1) not extracted from legacy applications but designed from scratch and (2) small and simple.

In Figure 11 we show the basic class structure of GPL. Details of some features are omitted. The architecture is simple: a *Graph* class holds the graph consisting of edges and vertices. Most features add a method to the *Graph* class and extend the *run* method used for benchmarks. Some features like *Number*, *Shortest*, or *Weighted* add fields for temporary values to *Vertex* and *Edge* and extend their *display* method to show the additional values. Features like *Connected* or *Cycle* also depend on methods implemented in previous features like search algorithms.

Feature	CI	CR	MI	Extended Methods
Prog	1	0	0	-
Benchmark	0	1	7	-
Directed	4	0	0	-
Undirected	4	0	0	-
BFS	2	1	3	Vertex.display
DFS	2	1	3	Vertex.display
Weighted	0	2	2	Edge.display, Edge.adjust
Transpose	0	1	1	-
Connected	1	2	1	Vertex.display, Graph.run
Cycle	1	2	1	Vertex.display, Graph.run
MSTKruskal	0	2	1	Vertex.display, Graph.run
MSTPrim	0	2	1	Vertex.display, Graph.run
Number	1	2	1	Vertex.display, Graph.run
Shortest	0	2	1	Vertex.display, Graph.run
StronglyC.	2	2	1	Vertex.display, Graph.run

CI: class introductions; CR: class refinements; MI: method introductions

Table 1: Feature Statistics for GPL.

Implementation with Mixin Layers. The granularity required to implement GPL is coarse. As illustrated in Table 1 most features introduce only new code fragments (25 methods and 21 fields into existing classes and 18 new classes). Methods were rarely extended (by means of method overriding). Of 84 methods in GPL only 4 were ever extended: the method *Vertex.display* was extended by 9 features, the benchmark method *Graph.run* was extended by 7 features and *Edge.display* and *Edge.adjust* were extended once. Not a single hook method was used to implement extensions at statement or expression level.

Even though all features could be implemented with the coarse granularity provided by the language, we found some code replication that could have been avoided if fine-grained extensions were available. Firstly, the mutually exclusive search features *BFS* and *DFS*, as well as *Directed* and *Undirected*, introduced similar methods that differ only in minor parts. Alternative implementations could use hook methods inside a common base feature to avoid code replication. In CIDE both variants can be combined in a single code base, in which both features only extend a single line as shown in Figure 12 (feature code is italicized and underlined respectively).

Secondly, the extension of the *Weighted* feature introduces a second *addAnEdge* method and a second *Edge* constructor with an additional parameter, though conceptually the *addAnEdge* method

[6]The second decomposition was also faster because we were already familiar with the source code. Still, the difference is significant.

```
1  public void search(Workspace w) {
2    int s = vertices.size();
3    if (s == 0) return;
4    for (int c = 0; c < s; c++)
5      vertices.get(c).init_vertex(w);
6    for (int c = 0; c < s; c++) {
7      Vertex v = vertices.get(c);
8      if (!v.visited) {
9        w.nextRegionAction(v);
10       v.dfSearch(w);
11       v.bfSearch(w);
12     }
13   }
14 }
```

Figure 12: Search method differs only in a single line.

and the constructor of the *Edge* class should be both extended with an additional *weight* parameter. So, with the *Weighted* feature enabled there were two *addAnEdge* methods, one with two and one with three parameters, even though only the method with three parameters should be used by clients.

Implementation with CIDE. In CIDE, we have composed the whole code base including several mutually exclusive features in a single code base, in which different code fragments are associated with features. This way, the file containing the code of the class *Graph* became fairly large (772 LOC), of which 88 % were associated with features. The projection facility came in handy to see only relevant code fragments.

We were surprised that despite GPL's apparent coarse granularity, we could still identify several situations, in which we benefit from CIDE's fine granularity and could reduce code replication. For example, we integrated both *search* methods that only differ in one line as shown in Figure 12. Similar code replication between the *Directed* and *Undirected* features could be avoided as well. Furthermore, we could implement the additional parameter for the *Weighted* feature much simpler without additional methods or constructors as shown in Figure 13. We simply associated all *weight* parameters with the *Weighted* feature (underlined). Because of reduced code replication and because we do not need implementation overhead of mixin layers, the code size of GPL in CIDE is 36 % smaller than in the original implementation (1,222 LOC instead 1,920 LOC).

```
1  class Edge {
2    Edge(Vertex start, Vertex end, int weight) {
3      this.start = start; this.end = end;
4      this.weight = weight;
5    }
6  }
7  class Graph {
8    Edge addAnEdge(Vertex start, Vertex end, int weight) {
9      Edge theEdge = new Edge(start, end, weight);
10     edges.add(theEdge);
11     start.addNeighbor(new Neighbor(end, theEdge));
12     return theEdge;
13   }
14 }
```

Figure 13: Parameter extension for addAnEdge.

6. RELATED WORK

SPL Adoption. Krueger distinguishes between three different adoption models for SPLs [28]. Firstly, the *proactive adoption model* relates to the waterfall approach of conventional software engineering. The SPL is planned up front and designed with all features in mind, therefore few carefully planned extension points are often sufficient and fine-grained extensions arise infrequently. Due to their coarse granularity, compositional approaches are well suited, though in our GPL case study we still discovered possibilities for improvement.

In contrast, the *extractive adoption model* converts an existing legacy applications to an SPL by decomposing its features – aiming to reduce adoption time, cost and risk. In legacy applications, features were not planned in the design phase and consequently dissolved in the implementation. Redesigning the complete legacy application is usually out of the question, because the additional effort could exceed the gained benefit of reduced time, costs, and risk. Extracting scattered feature implementations requires the ability to implement fine-grained extensions, so that annotative approaches and especially CIDE are better suited for SPL implementation. The Berkeley DB case study illustrates the need for fine-grained extensions when extracting features.

Finally, the *reactive adoption model*, which relates to spiral software engineering in conventional software, stands between the proactive and extractive adoption models. Not all features are foreseen and designed from scratch, but the SPL is extended in iterative steps. In this model, most extensions are preplanned, but it is not possible to design for extensibility in every case. So, we expect a medium to high number of fine-granular extensions as well.

This emphasizes the relevance of fine-grained extensions and the applicability of CIDE in SPLs beyond only decomposing legacy applications.

Related Tool Approaches. The concept of using the representation layer to show additional information without obfuscating the source code as in CIDE was already used by several development environments. Recent examples are *presentation extension* [12], *AspectBrowser* [14] and *Spotlight* [9].

IDEs for visual programming and intentional programming abstract from traditional code representations and store code in internal tree formats close to ASTs. The idea of storing program code in databases to allow flexible queries to create different views on the code goes back to Linton [31]. Modern examples are *Snippets* [54], *effective views* [17], and the *Domain Workbench* [43], that store all code in internal tree structures, similar to how features are assigned to code in CIDE.

There is also an impressive body of work on feature annotations of source code. However, such work usually does not aim at SPLs but at *virtual separation of concerns* to make concerns or features explicit. For example, Robillard and Murphy suggested *concern graphs* where developers can collect methods belonging to a feature in an external window [41]. Work on *visual separation of concerns (VSC)* extends this and provides aggregated views on the source code by features [8]. Furthermore, the *AspectBrowser* [14] and *JQuery* [16] use pattern expressions or queries to find code fragments belonging to a certain feature. Similarly to CIDE, feature annotations are stored externally, the code itself is not changed. While *FEAT*, *ConcernMapper*, *VSC*, and *JQuery* work at method level, thus providing only coarse granularity, *AspectBrowser* works on character level of unparsed code. In contrast to CIDE, these tools are used exclusively for source code exploration and navigation, whereas CIDE is designed as a software product line tool suite, that uses additionally annotations to configure the program.

Closest to our work on SPLs is an extended UML modeling environment suggested by Czarnecki and Pietroszek, in which users can assign individual model elements to features [11]. A tool uses this information to check whether all possible configurations are well-formed. However, while CIDE assigns only one or more fea-

tures to elements, they can assign arbitrary propositional formulas like 'entity X is only included if feature X is selected or feature Y is not'. Compared to CIDE, this raises the complexity for both the user interface and the back-end.

Cross-Section Views. Projections on features in CIDE resemble cross-section views, e.g., in 3D engineering or tomography. They hide all details unnecessary for the current task and let the user focus on certain details from different views. An early example of such cross-section views in software engineering is the concept of *program slicing* [53], in which the source code is projected to the fragments relevant for a certain control flow. Typically, it shows all code that can affect a selected variable and hides all other code. Program slicing helps abstracting from the whole program and to focus on a concrete, usually comprehension or maintenance focused task. Instead of projecting by control flow, CIDE projects (structurally) on individual features. This enables developers to understand programs by features and supports switching between different views dynamically.

Also the notion of on-demand remodularization follows the vision of cross-section views on a program. On-demand remodularization as introduced by Ossher and Tarr is the ability to extract a concern of an application into a new module without affection other concerns [38]. Typically, it is desired to decompose a program in one dimension first, e.g., data and classes, and later extract a concern in another dimension, e.g., features, which was not considered in the first decomposition. On-demand remodularization permits developers to identify and encapsulate new concerns at any time without necessity to rearchitect the software. This enables the developer to create only those modularizations that they need and as they need them [38].

Additionally to the virtual separation of concerns approaches introduced above, there are several approaches based on composition languages and tool support that physically remodularize the source code. An example is *effective views* [17] that are however limited to two dimensions, classes and 'modules'. Though modules are similar to features, they may not overlap. Another proposed tool for supporting remodularization in multiple dimensions is the *concern manipulation environment* that is aimed at describing and extract concerns [15]. Similarly, CIDE is a tool-based approach and allows to incrementally decompose a system into further features by associating code fragments with (potentially overlapping) features. Features can also be extracted in later development steps when needed. The key difference however is that CIDE aims directly at SPLs.

Feature Cohesion and Modular Reasoning. Implementing features modularly is a fundamental goal of compositional approaches. The motivation is that developers are able to understand and modify features in isolation without resorting to global reasoning or affecting other parts of the system if they are implemented cohesively [47]. There is a large body of work that discusses modular reasoning for different programming paradigms and languages, e.g., [46, 26]. Furthermore, Tarr et al. postulate that it should be possible to decompose a system in different dimensions and still implement all concerns modularly [49]. However, in our observation, such cohesive feature modules tend to become so complex that the benefit of modular reasoning diminishes in the presence of fine-grained extensions and with current compositional approaches.

In contrast, annotative approaches like preprocessors do not implement features cohesively. On the contrary, feature implementations are scattered throughout the system, preventing modular reasoning. To understand a feature, developers must search the feature code as a first step. However, approaches for virtual separation of concerns listed above like *concern graphs* support identifying and reasoning about scattered code. Although CIDE still provides exports into feature modules, it follows these ideas and provides tools to reason about scattered feature implementations instead of enforcing complex cohesive implementations.

Feature Model and Consistency.CIDE, as most other approaches for implementing extensions for features, does not deal directly with consistency constraints or use a feature model. A feature model on top of CIDE that describes the relationships between features, e.g., that one feature depends on another feature, is necessary for a holistic approach, but not relevant for the granularity discussion in this paper. Once features are related to each other, CIDE can check the feature association of all AST elements to ensure that every configuration is not only parseable, but also compilable. This requires that several constraints are fulfilled, e.g., that the target methods of all method calls are defined in all configurations. These checks follow the concepts of Czarnecki and Pietroszek [11] and Thaker et al. [50] but their discussion exceeds the scope of this paper.

7. CONCLUSION

There are many ways to implement features. However, when features have fine-grained extensions, as common when decomposing a legacy application, their implementations tend to become complicated, unreadable, and unmaintainable. Compositional approaches do not support fine-grained extensions, so that workarounds are required which raise the implementation's complexity. In contrast, annotative approaches can implement fine-grained extensions but introduce readability problems by obfuscating the source code.

To avoid these problems when developing SPLs with fine-grained extensions in a concise fashion, we built a tool, called CIDE, that simplifies SPL development. It is based on preprocessor semantics but uses background colors instead of source code statements and by providing the possibility to hide features it avoids obfuscating the code. CIDE restricts features to structural code elements in order to simplify usage for developers while still providing fine granularity. Finally, CIDE supports developers in understanding features with navigation and projection facilities and the possibility to export the SPL into distinct feature modules.

In two case studies we showed the advantage of CIDE over existing compositional approaches. It was possible to implement fine-grained features including statement and expression extensions and even signature changes without resorting to workarounds. With CIDE we could implement additional features that were not reasonably possible with compositional approaches and reduce code replication of earlier implementations.

In ongoing and future work, we add support for additional artifact types (e.g., C, C#, JavaScript, Grammars, XML) and formalize the criteria when an element can be colored. We intend to extend our export facility with additional target languages to analyze modular reasoning. Further, we plan to extend CIDE as a round-trip engineering tool where it is possible to edit the exported code and reimport it back to CIDE. This will eventually enable us to use CIDE to provide additional views on existing SPLs.

Acknowledgments. We thank Don Batory and Peter Kim for fruitful comments and discussions on earlier drafts of this paper and Marko Rosenmüller for sharing his experience with *'#ifdef'* preprocessors and some illustrations of hard to read C code exemplified in Figure 2.

8. REFERENCES

[1] S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *GPCE*, 2006.

[2] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. In *IEEE TSE*, 2008. Online first.

[3] S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner. An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706, University of Passau, 2007.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.

[5] I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. Working Conference on Reverse Engineering*. 2001.

[6] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Sci. Comput. Program.*, 53(3), 2004.

[7] A. Bryant et al. Explicit programming. In *AOSD*. 2002.

[8] M. Chu-Carroll, J. Wright, and A. Ying. Visual Separation of Concerns through Multidimensional Program Storage. In *AOSD*. 2003.

[9] D. Coppit, R. Painter, and M. Revelle. Spotlight: A Prototype Tool for Software Plans. In *ICSE*. 2007.

[10] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press, 2000.

[11] K. Czarnecki and K. Pietroszek. Verifying Feature-based Model Templates against well-formedness OCL Constraints. In *GPCE*. 2006.

[12] A. Eisenberg and G. Kiczales. Expressive Programs through Presentation Extension. In *AOSD*. 2007.

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

[14] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the Map Metaphor in a Tool for Software Evolution. In *ICSE*. 2001.

[15] W. Harrison et al. Concern modeling in the concern manipulation environment. In *ICSE Workshop on Modeling and Analysis of Concerns in Software*. 2005.

[16] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD*. 2003.

[17] D. Janzen and K. De Volder. Programming with Crosscutting Effective Views. In *ECOOP*. 2004.

[18] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. XVCL: XML-based Variant Configuration Language. In *ICSE*. 2003.

[19] S. Jarzabek and L. Shubiao. Eliminating Redundancies with a "Composition with Adaptation" Meta-Programming Technique. In *ESEC/FSE*. 2003.

[20] R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2), 1988.

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

[22] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *SPLC*, 2007.

[23] C. Kästner, M. Kuhlemann, and D. Batory. Automating Feature-Oriented Refactoring of Legacy Applications. In *ECOOP Workshop on Refactoring Tools*, 2007.

[24] G. Kiczales et al. Aspect-Oriented Programming. In *ECOOP*. 1997.

[25] G. Kiczales et al. An Overview of AspectJ. In *ECOOP*. 2001.

[26] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE*. 2005.

[27] G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In *ECOOP*. 2005.

[28] C. Krueger. Easing the Transition to Software Mass Customization. In *Intl' Workshop on Software Product-Family Eng.* 2002.

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

[30] T. Leich, S. Apel, and G. Saake. Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *ADBIS*, 2005.

[31] M. Linton. Implementing relational views of programs. *SIGPLAN Not.*, 19(5), 1984.

[32] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *ICSE*, 2006.

[33] R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *GCSE*. 2001.

[34] R. Lopez-Herrejon and D. Batory. Using AspectJ to Implement Product-Lines: A Case Study. Technical report, Dept. of Computer Sc., University of Texas at Austin, 2002.

[35] R. Lopez-Herrejon and D. Batory. Using Hyper/J to Implement Product-Lines: A Case Study. Technical report, Dept. of Computer Sc., University of Texas at Austin, 2002.

[36] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *OOPSLA*, 2001.

[37] G. Murphy et al. Separating Features in Source Code: an Exploratory Study. In *ICSE*. 2001.

[38] H. Ossher and P. Tarr. On the Need for On-Demand Remodularization. In *ECOOP Workshop on Aspects and Dimensions of Concerns*, 2000.

[39] R. Pawlak. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distrib. Sys. Onl.*, 7(11), 2006.

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

[41] M. Robillard and G. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *ICSE*. 2002.

[42] M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier. Avoiding Variability of Method Signatures in Software Product Lines: A Case Study. In *GPCE Workshop on Aspect-Oriented Product Line Engineering*, 2007.

[43] C. Simonyi, M. Christerson, and S. Clifford. Intentional software. In *OOPSLA*. 2006.

[44] Y. Smaragdakis and D. Batory. Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 11(2), 2002.

[45] H. Spencer and G. Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *USENIX Conf.*, 1992.

[46] R. Stata and J. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA*. 1995.

[47] W. Stevens, G. Myers, and L. Constantine. Structured Design. *IBM Systems Journal*, 13(2), 1974.

[48] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[49] P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*. 1999.

[50] S. Thaker et al. Safe Composition of Product Lines. In *GPCE*, 2007.

[51] S. Trujillo, D. Batory, and O. Diaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *ICSE*. 2007.

[52] A. Turon and J. Reppy. Metaprogramming with Traits. In *ECOOP*, 2007.

[53] M. Weiser. Program Slicing. *IEEE TSE*, 10(4), 1984.

[54] B. Westphal, F. Harris, and S. Dascalu. Snippets: Support for Drag-and-Drop Programming in the Redwood Environment. *Journal of Universal Computer Science*, 10(7), 2004.