Visualizing Software Product Line Variabilities in Source Code

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

Salvador Trujillo
IKERLAN Research Centre
Mondragon, Spain
STrujillo@ikerlan.es

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

Abstract

Implementing software product lines is a challenging task. Depending on the implementation technique the code that realizes a feature is often scattered across multiple code units. This way it becomes difficult to trace features in source code which hinders maintenance and evolution. While previous effort on visualization technologies in software product lines has focused mainly on the feature model, we suggest tool support for feature traceability in the code base. With our tool CIDE, we propose an approach based on filters and views on source code in order to visualize and trace features in source code.

1 Introduction

A *software product line (SPL)* is an efficient means to create a family of related programs for a domain in practice [3, 30]. Instead of implementing each program of this family from scratch, an SPL facilitates systematic reuse by modeling a domain with features (domain abstractions relevant for stakeholders, typically increments in functionality) and generating program variants from some assets that are common to the SPL [19, 3, 12].

Industrial SPLs typically have hundreds of features and large code bases of thousands or even hundreds of thousands of lines of code. A problem that is faced by many SPL implementation techniques is *tracing features* from the domain level (problem space) to their implementation (solution space) [12]. Often the implementation of a feature is scattered throughout several code units, for example in form of code fragments annotated with *#ifdef* directives. The lack of feature traceability causes several problems in development and maintenance [1], for example it is difficult to perform the following maintenance tasks for an individual feature:

- The specification of a feature changes: which code might be affected?
- A bug is reported in a certain feature: how to find and understand the feature's code?

We faced such problems, after decomposing a legacy application into features: From the Java and C versions of Oracle's embedded database engine Berkeley DB[1], we each decomposed several features to make it configurable as SPL [21, 34]. Berkeley DB (C version) was already configurable to some degree using *#ifdef* directives. However, when we tried to understand how some of the existing features were manifested in the source code, we needed to inspect the entire code base.

While there are several solutions to maintain feature traceability – e.g., architecture-based SPLs using frameworks or components [3], or specialized programming language concepts like feature-oriented programming [31, 4] – they are mainly used in academia. In practice simple solutions like conditional compilation with *#ifdef* or similar directives prevail.

Similarly, tool support and especially visualization techniques can help, but previous research on visualizing SPLs has mainly focused on the feature modeling and the product derivation process, e.g., [6, 35, 32]. In current visualization approaches the SPL's implementation is usually not considered.

In this paper, we discuss different approaches to support developers in *understanding and exploring individual features* by combining several visualization techniques. Specifically, we propose virtual views on the source code, depending on a selection of features. We base our implementations on a tool for implementing SPLs called *Colored Integrated Development Environment (CIDE)* which we presented in earlier work [22]. Although CIDE is a proprietary tool which maintains a direct mapping between features and their (possibly scattered) implementation, the presented concepts could easily be adapted to support existing SPLs that already used *#ifdef* or similar concepts.

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

2 SPL Implementation Approaches

There are many approaches to SPL implementation. Most of them can be categorized either as compositional or as annotative approach [22].

Compositional Approaches. Compositional approaches implement features as distinct (physically separated) code units. To generate a product line member for a feature selection, the corresponding code units are determined and *composed*, usually at compile-time or deploy-time. There is a large body of work on feature composition, usually employing component technologies [38], or specialized architectures and languages like frameworks [18], mixin layers [36, 2], AHEAD [4], multi-dimensional separation of concerns [39], and aspects [25]. Depending on the concrete approach or language, the composition mechanism varies from assembling plug-ins to complex code transformations, but the general idea of composition as illustrated in Figure 1 is the same.

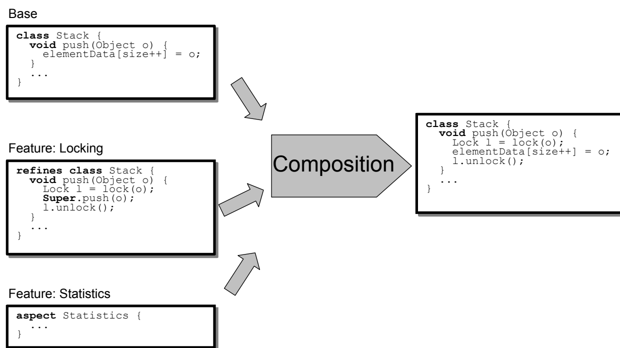

Figure 1. Composing code units.

The advantage of compositional approaches is that they provide a direct link between a feature and its implementation. Thus, they achieve a high degree of feature traceability. A code unit (e.g., a plug-in or a mixin layer) can be associated directly with a feature. Feature code can be found directly in the code units associated with a feature, it is not intermixed with code from other features.

However, compositional approaches share the problem that they are unable to implement SPLs at a fine granularity [22]. For example, they do not support a feature that needs to change a single line of code in another feature. Instead they usually build on a rather coarse-grained architecture. Furthermore, many of them require specialized languages (breaking the tool chain) or architectural overhead. If used for legacy applications, they force a new paradigm or architecture upon existing structures. Therefore, although compositional approaches are popular in academia, they are hardly used in industrial projects so far.

Annotative Approaches. In contrast, annotative approaches implement features with some form of explicit or implicit annotations in the source code. The prototypical example, which is commonly used in industrial SPLs are *#ifdef* and *#endif* statements of the C preprocessor to surround feature code. Such techniques are also common in commercial SPL tools as *pure::variants* [5] or *Gears* [27]. Other examples of annotative approaches are *Frames/XVCL* [17], *explict programming* [8], *Spoon* [29], *software plans* [11], *metaprogramming with traits* [40], and *aspects using annotations* [26].

Annotative approaches all share the problem of feature traceability. The implementation of a feature is typically scattered throughout several code units. In some approaches, features are partly modularized leaving just some 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. In Figure 2, we show a shortened excerpt from Berkeley DB in which the preprocessor is used to achieve variability. Despite the problem of lacking feature traceability (and others not relevant for this paper [37]) annotative approaches are commonly used in industry because they are a simple technique without much overhead and because they are less intrusive with the current design and development process [10].

```
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.

In this work, we will address the problem of lacking traceability in annotative approaches with additional tool support. Our aim is to close the gap between compositional and annotative approaches by introducing visualization techniques. Specifically, we introduce views on specific features, which enable developers to explore a feature's code as if it was physically separated. The views support code exploration and enhance feature traceability.

3 Virtual Separation of Concerns

In this paper, we build on the concept of *virtual separation of concerns* [22]: concerns (and thus including also features) are not physically separated but just annotated as in annotative approaches. However, these annotations are managed and controlled by a tool infrastructure. This tool infrastructure enables novel concepts of virtual views on the source code for source code navigation and exploration.

The concept of virtual separation of concerns roots back to our experience on decomposing legacy applications [21]. For this task, we needed to analyze how feature code was scattered and tangled in a legacy implementation. In early discussions, we literally used colored pens on printouts of code fragments to mark feature code, one color per feature. As this turned out conveniently, we created a tool called *Colored Integrated Development Environment (CIDE)* to facilitate this otherwise manual approach.

CIDE is built on top of the Eclipse's Java development environment. Developers can select fragments of the code and assign a feature to it. Following the metaphor of pens on paper, CIDE does not add *#ifdef* or similar directives to the source code, but uses the representation layer. It shows the annotated code fragment with a background color that represents the feature, see the example in Figure 3. All annotations are managed within the tool, which enables new possibilities for visualizations. CIDE is primarily used as SPL tool; to generate a variant those code fragments associated with unwanted features are removed.

As *#ifdef* directives can be nested to specify glue code between two features (that is only included when both features are selected), also colors in CIDE can overlap. If two or more features are annotated to a code fragment, this is considered as glue code between those features and only included in a generated variant if all annotated features are selected. Typically such glue code is annotated in a nested form, e.g., the entire class is 'blue' and inside one method or statement is also 'red'. In the editor, the background colors of all involved features are blended; we will discuss implications of this later.

Underlying Structure. A concept that is necessary to understand how CIDE works and which we later use to implement our views, is that CIDE enforces disciplined annotations based on the underlying code structure. A developer cannot annotate arbitrary code fragments – possibly delimited by offset and length – but only structural elements of the code, e.g., classes, methods, statements or even parameters. The structure is determined internally by the code's *abstract syntax tree (AST)* as depicted in Figure 4. When a code fragment is selected, this selection is internally mapped to the according AST nodes. In our example the underlined code is mapped to the grayed AST elements.

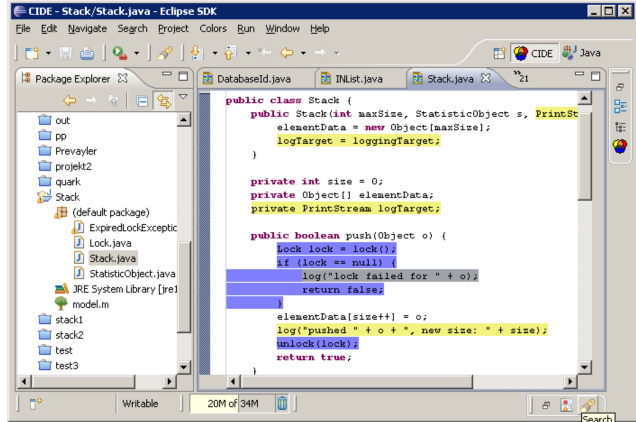

Figure 3. CIDE Screenshot

```
1 class C {
2   void m(int p){
3     s1();
4     s2(p,true);
5   }
6   void p() {}
7 }
```

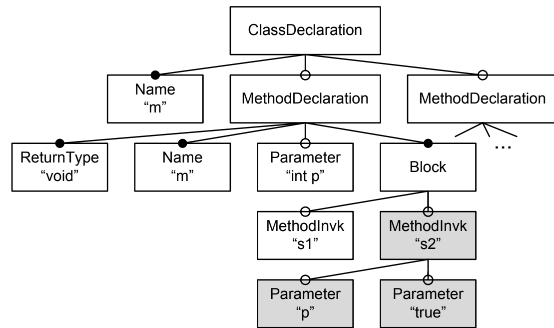

Figure 4. AST Example.

In an AST, we can distinguish optional nodes and mandatory nodes. Optional nodes can be removed without invalidating the syntax. In Figure 4, we use the FODA notation [19] and connect optional AST nodes with an empty circle and mandatory nodes with a filled circle. The class node is optional, so are the method nodes, or the nodes for method calls and parameters. In contrast the node for return type or block are mandatory; removing only the return type of the method would cause an invalid syntax. In CIDE, only optional AST nodes can be annotated.

Mapping annotations to underlying structures has two benefits. First, only disciplined annotations are allowed, which avoids many potential errors like annotating only an opening but not the closing bracket. These disciplined annotations distinguish CIDE from common preprocessors. Second, the underlying structure can be used for code manipulations and visualizations. For example, a variant of the

SPL is generated by AST transformations, so developers do not have to deal with pure syntax elements like the separating comma between parameters [22].

In the following, we discuss four concepts used by CIDE to support feature traceability and code exploration in SPLs though visualization: scaling, views on the file system, and two forms of views on file content.

3.1 Scaling

In previous work, we have only considered annotating code fragments inside a file. Inside files, using the underlying AST scales well: It is possible to annotate entire classes, entire methods, individual statements or even parts of expressions or parameters.

However, for CIDE to scale for large SPLs, we also need to consider coarser granularities than file fragments. Often entire files or even packages (directories) can belong to a feature. For example, in Berkeley DB (Java version) the transaction functionality is already modularized to a large degree in a package. There are still several scattered calls to the transaction subsystem, however the core implementation is located in few files. The traditional approach (also for *#ifdef* preprocessors) would be to annotate the entire content of all these files. However, this makes locating feature code more difficult than necessary, because a developer first has to look inside these files.

Instead, we add the possibility to annotate files and directories directly. To maintain the color metaphor, we also add background colors to files and directories in Eclipse's standard project explorer view as shown in Figure 5. For developers it becomes thus straightforward to recognize that an entire directory belongs to a feature. This way the color metaphor scales from smallest code fragments within a file up to entire directories.

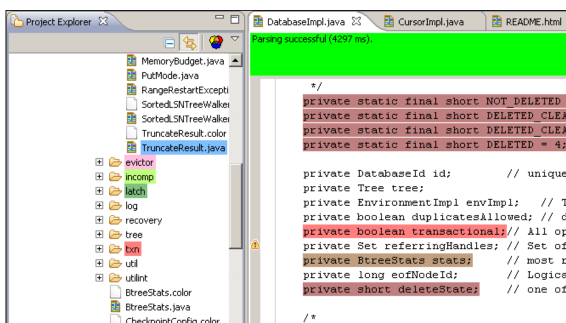

Figure 5. Colors scale from directories to code fragments

3.2 Views on the File System

When entire files or directories are colored a user can recognize features without looking into all files. However, there are still two problems. First, when many files are involved it can still be tedious to find all files in the directory structure. Second, when only a code fragment inside a file is annotated – instead of an entire file – the developer still won't find all feature code without looking into all files.

To address these problems, we introduce a filter functionality that creates a *view on the file system*. In order to use this, the developer has to select one or more features from the feature model (or a list of features) and press a new filter button in the standard Eclipse project explorer. When activated only those directories or files are shown in the project explorer which contain a file or code fragment annotated with one of the selected features. Thus, the developer can trace a feature to all its code fragments.

Note, to select features for a view, a simple list of features without further dependencies is sufficient. Although it is possible to create views on invalid configurations this way, e.g., show two mutually exclusive features at the same time, this flexibility might be necessary for certain tasks. For example, it must be possible to create a view on a single feature without first having to create a valid configuration (which might require several other features). Dependencies between features are only required for the derivation process, but not necessarily to provide views for developers. Nevertheless, CIDE shows an indicator whether the current selection would also be a valid configuration.

In Figure 6, we show how this is implemented in CIDE: in the upper 'Project Explorer' view there is a new filter button. When activated only those files that contain code fragments that are annotated with at least one of the selected features are shown. Features can be selected from the lower view ('Feature List'); when the selection is changed the view on the file system is updated instantly. For example, when selecting the transaction feature for Berkeley DB SPL and activating the filter function (as shown in Figure 6), only those files with some transaction code are shown, all others are hidden. It does not only show the 'txn' package which is entirely colored because it contains only code of the transaction feature, but it also shows files from other packages that contain annotated calls to the transaction system.

3.3 Views on File Content

The filter function is very useful to create views on the file system structure. However, it only shows which files contain annotated code fragments. When opening a file in the editor, we still have to search for all locations. Compared to the length of the entire file, the annotated code fragment can be relatively small, e.g., only a single statement or parameter.

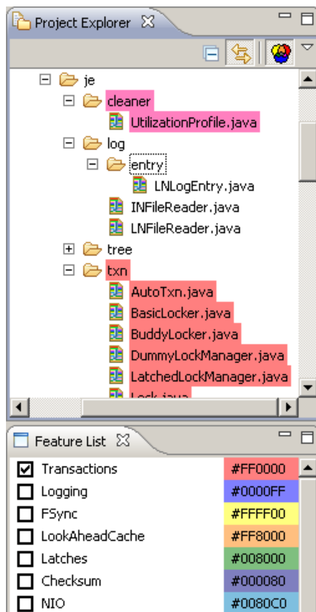

Figure 6. A filter creates a view on the file system

Therefore, we also provide views on the source code *inside a file*.

We experimented with several different versions how to provide *views on a file*. We finally implemented two different solutions. Interestingly, these views correspond closely to the classification by Heidenreich et al. [14]. The first hides all features which are not selected and realizes a *variant view*, i.e., it shows how a certain variant looks like. The second solution shows all code of selected features but hides everything else except some context code that is necessary understand the location of the code fragment. It can be classified as *realization view* that shows the code fragments realizing a feature [14].

Variant View. Our first implementation allowed hiding all code fragments that were annotated with any features not currently selected. This view is very similar to the actual result of the variant generation process. Consider an example, where the user selects only the transaction and the logging feature and opens a file which contains a some transaction and logging code fragments (e.g., to lock some statements and log results). In this example, the user sees only the base code that is not annotated and the transaction and logging code. All remaining code that is annotated with other features is hidden. Still, hidden code fragments are indicated by a marker (in the color of the hidden code fragment), as illustrated in Figure 7 for a simple example of a Stack class with two features. Given that the selected features are a valid configuration of the feature model, the code in this view represents a complete variant and could even be compiled.

The functionality to hide code is implemented using Eclipse's projection framework, which commonly is used to fold methods or comments in the Java editor or other editors. In contrast to code folding, hiding feature code also works with code fragments inside a line (code folding allows only to hide entire lines) and a symbol is shown to indicate the hidden code.

This view can also be used to not only view a variant, but also to trace a single feature. Therefore, the user would only select a single feature and would only see the base code and this feature. The advantage of this approach is that the 'uncolored' code which surrounds the feature code is still present, so it is easy to understand how the feature code interacts with the SPL. However, at the same time, this is also a disadvantage, because on most files there is much 'uncolored' code so that it might still be tedious to search the potentially small fragments of feature code. Furthermore, a second problem is that code that is annotated with two or more features (overlapping features for glue code) is only shown when all annotated features are selected for the view. For example, 'red code' inside 'blue code' is only shown when 'blue' *and* 'red' are selected, but hidden when only 'red' is selected. This might not fit the intuition if a developer wants to see 'all transaction code'.

Realization view. The second implementation of a view on file content addresses the limitations of the first when it comes to tracing a single feature. First, it does not show all the 'uncolored' code, but only those code fragments necessary to understand the current location. Second, it shows the entire code annotated with a feature, even if it is annotated with further features.

This view is more difficult to implement, because we need to decide which code to hide. For example, when only a statement in a method is annotated, we cannot hide all the remaining code, because the developer would not know in which class or method this single statement is located (potentially even in a method in an anonymous inner class or inside a static initializer). This means that we need to show some context for each location. Determining the context is the difficult decision and there may be many different solutions.

Using the distinction between optional and mandatory nodes in the underlying structure, we found one approach that provides a satisfactory solution. Actually, given the underlying structure, our algorithm to determine the necessary context is fairly simple. First, all code fragments annotated directly with the selected features are shown. Second, all their parents are shown as well (recursive to the root of the AST). Third, from every shown node all *mandatory child nodes* are shown, but not the optional ones.

Let's illustrate this algorithm on our initial example in

```
public class Stack {
    public Stack(int maxSize, StatisticObject s, PrintStream loggingTarget) {
        elementData = new Object[maxSize];
        logTarget = loggingTarget;
    }

    private int size = 0;
    private Object[] elementData;
    private PrintStream logTarget;

    public boolean push(Object o) {
        Lock lock = lock();
        if (lock == null) {
            log("lock failed for " + o);
            return false;
        }
        elementData[size++] = o;
        log("pushed " + o + ", new size: " + size);
        unlock(lock);
        return true;
    }

    public Object pop() {
        Lock lock = lock();
        if (lock == null) {
            log("lock failed for pop");
            return null;
        }
        Object r = elementData[--size];
        unlock(lock);
        return r;
    }

    private void log(String msg) {
        msg = "Stack Debug Output: " + msg;
        logTarget.println(msg);
    }

    private Lock lock() {
        assert Lock.freeLock() > 0;
        Lock existingLock = Lock.findLock(this);
        if (existingLock == null) {
            Lock newLock = new Lock(this);
            return newLock;
        } else {
            return existingLock;
        }
    }

    private void unlock(Lock lock) {
        assert lock.getTarget() == this;
        try {
            lock.unlook();
        } catch (ExpiredLockException e) {
            lock.rollback();
        }
    }
}
```

```
public class Stack {
    public Stack(int maxSize, StatisticObject s, PrintStream loggingTarget) {
        elementData = new Object[maxSize];
        logTarget = loggingTarget;
    }

    private int size = 0;
    private Object[] elementData;
    private PrintStream logTarget;

    public boolean push(Object o) {
        elementData[size++] = o;
        log("pushed " + o + ", new size: " + size);
        return true;
    }

    public Object pop() {
        Object r = elementData[--size];
        return r;
    }

    private void log(String msg) {
        msg = "Stack Debug Output: " + msg;
        logTarget.println(msg);
    }
}
```

Figure 7. View on 'yellow' Logging feature hides all 'blue' Transaction code.

Figure 4. The second method call in the first method is shown because it is annotated. Its parent the 'block' node is shown. The first statement in the block is optional, therefore it is not shown. The block's parent – the method *m* – is shown, including is mandatory children the return type and name of the method, but not including the optional parameters. Also the method's parent, the class is shown, again not including the optional children like the other method. The resulting view is depicted in Figure 8, it contains neither the unnecessary method, nor the method's parameter, nor the other statement.

A view created with this algorithm is not complete and could not be compiled. It is much closer to a mixin [7] or partial class in C#, as used in compositional approaches. Only the necessary context to understand an individual feature is shown, not an entire compilable variant. Again, the difference between feature code and context becomes obvious by the annotation shown as background color.

```
1 class C {
2   void m(){
3     s2(p,true);
4   }
5 }
```

Figure 8. View shows feature and context

In larger files with much uncolored code, the difference is even more significant. In Figure 9, we show another example, this time from an (excerpt of an) ANT build script (XML). The annotated code fragment is only a single line in a long XML file. In XML files element names are mandatory but attributes are optional in the underlying structure, thus the view contains only the necessary elements without attributes as context.

We found that views created with this algorithm are usually suitable to show enough context to roughly understand how the annotated code fragments related to the remaining code. Still, some fine-tuning is possible which could be offered as options to the users. For example, it would also be possible to hide only *entire lines* that do not contain any visible code. In this case some more context information like parameters in methods are visible without making the view longer in terms of lines of code. Alternatively, we could show a fixed frame of *n* lines or statement as context before or after each fragment of feature code, as in the Unix *grep* utility. Furthermore, again markers can be shown in the code to indicate hidden fragments.

3.4 Summary

To summarize, with views we virtualize the SPL code in CIDE. Colors are used consistently to represent features, scaling from entire directories or files to code fragments of different sizes inside files. While the feature code may

```
1  <project name='Release G.'>
2    <description> ... </description>
3    <patternset id="tool.patterns">
4      <include name="modelexplorer*"/>
5      <include name="ant/**/*.jar"/>
6      <include name="applybali2jak*"/>
7      <include name="xak*"/>
8      <include name="infozone*"/>
9      <include name="saxon*"/>
10   </patternset>
11   <target name='ahead' ...>
12     <antcall inheritall='false' ...>
13         <param name='dir.source' ... />
14     </antcall>
15   </target>
16 </project>
```

```
1 <project>
2   <patternset>
3     <include name="xak*"/>
4   </patternset>
5 </project>
```

Figure 9. View on XML fragment and context.

still be scattered across the entire code base, there are several mechanisms to create a view for one or more selected features in order to trace features directly to their code fragments. First, only relevant files are shown in the directory structure of the project, second, also inside files only relevant parts are shown. This way, feature traceability as known from compositional approaches can be emulated even for SPLs that are developed with common annotative approaches.

4 Discussion

Already in previous work, we used CIDE in a series of case studies of different size and different languages, e.g., a small Java SPL for graph algorithms [22], Berkeley DB [22], documentation and build scripts from the AHEAD tool suite, SQL grammars, Haskell programs, and an industrial SPL written in C [23].[2] We do not want to repeat these case studies, however the new possibility to create views raises several issues that we discuss in the following.

Using colors to represent features. The decision to use background colors to represent features instead of additional language constructs like *#ifdef* has been controversial. There are several issues which have to be considered.

On the positive side, colors are intuitive to map. In most editors background colors are not already used to represent some other information (in contrast to different text styles like bold or italics or foreground colors). In contrast to directives like *#ifdef*, background colors do not obfuscate the source code or influence the layout of the source code.

However, humans are not able to distinguish and recognize many colors clearly. It would not be possible to map a feature A to light red and another feature B to a slightly darker red, as developers would constantly confuse them. It is not possible to distinguish more than a handful of features by colors, which is not enough for the majority of SPLs. Even worse, handicapped users – e.g., red-green color blindness is quite common affecting 7–10 % of all males – can distinguish even less colors.

Therefore, it is important to emphasize that colors are not mapped uniquely (1:1) to features. Already when we colored features with pens on printouts, we noticed that there are rarely more than three or four colors on a single sheet of code. The same applies for editors: on a single screen only few colors appear at the same time. Therefore, we found that a repeating list of 12 different colors is sufficient to represent even large feature models. Additionally the developer may change the colors for all features. Users do not need to recognize features from the used colors. Instead they use colors to determine where feature code begins and ends and how features are tangled. The concrete names of features annotated at a code fragment can usually be inferred from the context or looked up from a tool tip in CIDE.

This way, it is even possible to represent overlapping colors for glue code by blending the involved colors. For example, when a class is colored in blue and a method within is additionally colored in red; the overlapping section is represented as purple. Again it is only possible to see where the overlapping sections start and end, but the involved features can be looked up from a tool tip. Therefore, blending colors does not pose a limitation to recognizing features.

Integrating existing preprocessors. In our current implementation CIDE differs from existing preprocessors in that annotations are mapped to the underlying AST and annotations are not shown with additional keywords but on the representation layer. There are two ways how annotations can be stored persistently. In our implementation feature annotations are also stored separately (for every file there is

[2] CIDE, together with case studies from this and earlier publications are available on the project's web site http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/.

a '.color' file that contains the annotations). Alternatively, it would be possible to store annotations inside the code using special directives (in the simplest case just with *#ifdef*), but erase them when loaded into the editor.

The first solution of external storage has the advantage, that existing legacy code must not be modified. Especially in industry during early adoption of SPL technology this is an advantage, because the first SPL implementations do not interfere with ongoing development. However, this approach binds all developers to an IDE which understands these annotations and updates them when the code is edited. It is no longer possible to edit the source code in an external tool (e.g., *notepad* or *vi*) without the possibility to loose or move annotations.

In contrast, the second solution makes invasive changes to the source code. When a code fragment with annotations is saved, these annotations are stored using additional directives. When loaded again, these directives are parsed, but not shown to the user. In the editor, again background colors are used. The advantage of this approach is, that annotated code can still be viewed and even edited with external editors. However, this also allows users to change annotations with external editors which can result in inconsistent annotations, e.g., annotating code fragments that do not correspond to optional AST nodes. This would prevent loading the file in CIDE or generating a variant until the annotation is fixed to its 'disciplined' form.

The advantage of the second solution is, that CIDE can also be used as visualization techniques for projects which already use some existing preprocessor mechanism for variability (e.g., the C preprocessor, Frames, XVCL, pure::variants, or Gears). As long as these preprocessor instructions are used in a disciplined way, CIDE can also be used to provide views for those projects. The additional instructions are hidden and replaced by colors. All presented filters and views can be used on these projects.

Consistent and editable views. While read-only views are certainly helpful to explore features in an SPL, editable views are even more helpful as developers can directly make changes in the view without first going back to the original code base. This raises a question of consistency: if code is inserted while some code is hidden, how does the editor know where to put this new code?

At this point, the markers that indicate hidden code are helpful. The markers show where code is hidden, and a developer can clearly make changes before or after this marker. When developers delete a code fragment, these markers also clearly indicate that they delete some hidden code. This way consistency is ensured, while it is still possible to edit views.

Discipline to annotate entire feature. The idea of adding tool support or special annotations to navigate code is not new. For example, in the FEAT tool [33] the developer manually adds methods or classes to a concern model and this model can later be used to navigate and explore these concerns. Similarly, AspectBrowser [13] and JQuery [15] allow to explore the code with queries that can be created for certain concerns or features. However, a problem of these approaches is that there is no incentive for developers to update the models or queries in these tools. There is no incentive to add *all* code fragments that belong to a certain feature. However, when these models are not updated or complete the tools are of limited use.

In CIDE this is different. Annotations in CIDE are not just used for code exploration and understanding. Instead, annotations are primarily used to generate variants in the SPL, so that there is a strong incentive to have annotations updated and complete. Completeness is simply ensured by the fact that generated variants are compiled and tested in the normal development process. CIDE's visualization capabilities are just an additional possibility to use these already existing annotations in order to support feature traceability and code exploration.

In a separate line of research, we also focus on how we can support the developer in checking correctness and completeness of annotations. For example, we extended Java's type system to check that every variant is type-safe, which already detects many potential inconsistent or incomplete annotations [20, 23]. The feature location problem, i.e., to find features in legacy code, is another area of research which we want to address with CIDE in future work.

5 Related work

CIDE itself was presented in recent work. We discussed granularity (inside a file) and the mechanism to use the underlying AST [22]. We discussed how to extend CIDE and the underlying structure to other languages including C, C#, JavaScript, ANTLR, Haskell, and XML [23]. Finally, we also formalized how to type-check entire SPLs based on their annotations [20]. In contrast, in this paper we focused only on the novel contributions of visualization using filters and views.

Closely related to CIDE are tools to explore and query legacy applications, most notably FEAT [33], AspectBrowser [13], *JQuery* [15], and *Spotlight* [11]. The problem of most of these approaches is that an internal model – which describes which code fragments belong to which concerns or features – has to be manually maintained or specified by an adequate query. As such information is maintained separately from the source code, there is little incentive to update these models or queries. In contrast, CIDE uses annotations that are already provided in order to generate variants of an SPL.

Next, there are several approaches that create views on

the source code. A recent and very popular approach is Mylyn [24], an Eclipse plug-in that creates task-based views on the source code, most notably on the file system. Depending on the task, only relevant files are shown in the project explorer. Effectively, Mylyn provides a view on the file system which is based on the context of the current task (which is collected in an internal model from development activity). CIDE's view on the file system was inspired by Mylyn, but uses feature annotations instead of the task context model. While Mylyn's model also includes information about classes and methods, views on file content in the editor are provided only in a basic form. Mylyn uses Eclipse's code folding capabilities: all methods which are not in the current context are automatically folded.

Beyond views on the file system, there are several approaches to provide views on the content of a file. Early approaches reach back to the ideas of program slicing by Weiser [41], i.e., read-only views showing only relevant code for a certain control flow, and relational views using an underlying database by Linton [28]. More recent approaches are *visual separation of concerns* [9] and *effective views* [16], which transform the underlying code for different views, however both are confined to a certain underlying model. For example, effective views are only provided for a confined specialized programming language to ensure safe transformations and editable views. CIDE's views on files that just hide parts of the code are simpler and can be supported for arbitrary languages, as long as there is an underlying structure [23].

Finally, in parallel work, Heidenreich et al. [14] introduced the notion of variant views and realization views and implemented them in their tool *FeatureMapper* which is similar to CIDE. However, FeatureMapper focuses on tracing features to model elements (e.g., classes or associations in an UML class diagram) instead of code fragments, thus the distinction between annotating files and annotating file content is not necessary. Interestingly, in some views FeatureMapper also uses colors to represent features in those models.

6 Conclusion

Implementing software product lines is a challenging tasks. Especially with annotative approaches, e.g., using *#ifdef* directives, which are commonly used in industry, the code that implements a feature can be scattered across several code units. This makes tracing features difficult, although it is essential to develop or maintain SPLs.

In this paper, we discussed several extensions to a tool based annotative approach called *CIDE* for implementing SPLs. As with preprocessors code fragments are annotated with features and removed in order to create variants. However, we have shown how to use these annotations beyond only variant generation. With them, we provide views on the source code to overcome the feature traceability problem. When developers selects one or more features they are interested in, only the relevant files that contain some feature code are shown in the project explorer (view on the file system). When the developers then open a file, again only the feature code and necessary context information are shown (view on a file).

With these views, features can be traced directly to their implementation. This approach is supported by the fact that all views are editable and scale from coarse-grained features implemented by entire packages or files, to features that are implemented by small scattered code fragments (fine-grained). If existing annotations using other preprocessor technologies already exist and they have been used in a disciplined form, CIDE can even be extended to provide views for these existing SPLs.

In future work, we intend to evaluate our approach in an empirical study. Furthermore, we will evaluate different means to create views on file content empirically, in order to determine which amount of context is required to understand feature code in isolation.

References

[1] G. Antoniol, E. Merlo, Y.-G. Guéhéneuc, and H. Sahraoui. On feature traceability in object oriented programs. In *Proc. ASE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 73–78, 2005.

[2] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.

[5] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352, 2004.

[6] G. Botterweck, D. Nestor, A. PreuSSner, C. Cawley, , and S. Thiel. Towards supporting feature configuration by interactive visualisation. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, pages 125–131, 2007.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 303–311, 1990.

[8] A. Bryant et al. Explicit programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 10–18, 2002.

[9] M. Chu-Carroll, J. Wright, and A. Ying. Visual separation of concerns through multidimensional program storage. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 188–197, 2003.

[10] P. Clements and C. Krueger. Point/counterpoint: Being proactive pays off/eliminating the adoption barrier. *IEEE Software*, 19(4):28–31, 2002.

[11] D. Coppit, R. Painter, and M. Revelle. Spotlight: A prototype tool for software plans. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 754–757, 2007.

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

[13] W. Griswold, J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 265–274. 2001.

[14] F. Heidenreich, I. Şavga, and C. Wende. On controlled visualisations in software product line engineering. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, 2008.

[15] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 178–187, 2003.

[16] D. Janzen and K. De Volder. Programming with crosscutting effective views. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 195–218. 2004.

[17] S. Jarzabek et al. XVCL: XML-based variant configuration language. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 810–811, 2003.

[18] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

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

[20] C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. 2008.

[21] C. Kästner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 223–232, 2007.

[22] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 311–320, 2008.

[23] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-independent safe decomposition of legacy applications into features. Technical Report 2, School of Computer Science, University of Magdeburg, Germany, 2008.

[24] M. Kersten and G. Murphy. Using task context to improve programmer productivity. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 1–11, 2006.

[25] G. Kiczales et al. Aspect-oriented programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. 1997.

[26] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. 2005.

[27] C. Krueger. Easing the transition to software mass customization. In *Proc. Int'l Workshop on Software Product-Family Eng.*, pages 282–293, 2002.

[28] M. Linton. Implementing relational views of programs. *SIGPLAN Not.*, 19(5):132–140, 1984.

[29] R. Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distrib. Sys. Onl.*, 7(11):1, 2006.

[30] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[31] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. 1997.

[32] R. Rabiser, D. Dhungana, and P. Grünbacher. Tool support for product derivation in large-scale product lines: A wizard-based approach. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, pages 119–124, 2007.

[33] M. Robillard and G. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 406–416, 2002.

[34] M. Rosenmüller et al. Fame-dbms: Talor-made data management solutions for embedded systems. In *Proc. EDBT Workshop on Software Engineering for Tailor-made Data Management*, pages 1–6, 2008.

[35] D. Sellier and M. Mannion. Visualizing product line requirement selection decisions. In *Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE)*, pages 109–118, 2007.

[36] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[37] H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C news. In *Proc. USENIX Conf.*, pages 185–198, 1992.

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

[39] P. Tarr et al. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 107–119, 1999.

[40] A. Turon and J. Reppy. Metaprogramming with traits. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 4609 of *Lecture Notes in Computer Science*, pages 373–398. 2007.

[41] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, 1984.