Code Generation to Support Static and Dynamic Composition of Software Product Lines

Marko Rosenmüller, Norbert Siegmund, Gunter Saake
School of Computer Science, University of Magdeburg, Germany
{rosenmue,nsiegmun,saake}@ovgu.de

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

Abstract

Software product lines (SPLs) are used to create tailor-made software products by managing and composing reusable assets. Generating a software product from the assets of an SPL is possible statically before runtime or dynamically at load-time or runtime. Both approaches have benefits and drawbacks with respect to composition flexibility, performance, and resource consumption. Which type of composition is preferable should be decided by taking the application scenario into account. Current tools and languages, however, force a programmer to decide between static and dynamic composition during development. In this paper, we present an approach that employs code generation to support static and dynamic composition of features of a single code base. We offer an implementation on top of FeatureC++, an extension of the C++ programming language that supports software composition based on features. To simplify dynamic composition and to avoid creation of invalid products we furthermore provide means to (1) validate the correctness of a composition at runtime, (2) automatically instantiate SPLs in case of stand-alone applications, and (3) automatically apply interaction code of crosscutting concerns.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors—Code Generation, Preprocessors

General Terms Design, Languages

Keywords Software product lines, feature-oriented programming, static feature binding, dynamic feature binding

1. Introduction

Software product lines (SPLs) provide means to compose software products that match the requirements of different application scenarios from a single code base. SPLs can be developed using a variety of implementation techniques. Well known concepts are preprocessor definitions, components, collaboration-based designs, *aspect-oriented programming (AOP)* [25], *feature-oriented programming (FOP)* [36, 8], and *aspectual feature modules* [3]. A main difference between implementations of these approaches is the type of composition used to create a concrete product or *SPL instance*. Some of them support static composition of program code at compile time or in a preprocessing step and others support dynamic composition, e.g., using components, at application startup or at runtime.

Dynamic composition provides a certain flexibility by allowing a programmer to select the needed functionality at runtime. For example, loading only required functionality on a mobile device from a network according to the underlying hardware or user preferences can reduce network load and avoids the necessity to provide any variant of a program. Unfortunately, dynamic composition has typically a negative effect on performance because it introduces an overhead. In contrast, static composition avoids such an overhead needed to support dynamic composition and enables optimizations on the source code level (e.g., function inlining) [21, 12]. However, a static approach is not as flexible as a dynamic approach since the functionality of a software product has to be known before deployment. Hence, both compositional approaches are useful for different application scenarios but the concrete application scenario might not be known until deployment.

Using current implementation techniques, the developer of an SPL is forced to choose between static and dynamic composition at development time. Changing the type of composition at deployment time is only supported by a few approaches [18, 12]. As a result, source code developed for static composition cannot be easily reused for dynamic composition and vice versa. In this paper, we present an approach that supports static and dynamic composition of features of a single code base which allows us to choose the type of composition at deployment time. While static composition is usually supported by tools [6, 37, 9] dynamic composition of features can be a complex task. This includes manual instantiation of features and has to cover crosscutting concerns, feature interactions, and the validation of correct SPL instantiation. To solve this problem we provide an infrastructure for dynamic instantiation and validation of SPLs.

Our approach is based on FOP which allows us to generate SPL instances by composing modularized features. We have implemented the presented approach using *FeatureC++*[1], an FOP extension for the C++ programming language [2]. We provide means to:

- compose features of an SPL statically or dynamically from a single code base,
- automate dynamic instantiation of SPLs with an extensible approach,
- apply code for feature interactions automatically,
- validate the correctness of an SPL configuration at runtime.

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

GPCE'08, October 19–23, 2008, Nashville, Tennessee, USA.

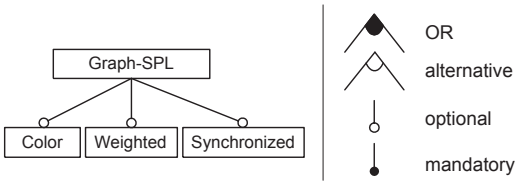


Figure 1. Feature diagram of a graph product line with optional features COLOR, WEIGHTED, and SYNCHRONIZED.

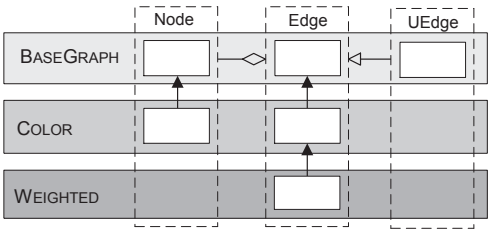


Figure 2. Decomposition of classes (vertical bars) with respect to features (horizontal bars) in feature-oriented designs.

2. Background

An SPL is used to create similar programs that share some common *features*. A feature of an SPL represents a functional requirement on a software that is of interest to some stakeholder [8]. SPLs can be described using *feature models* [22], e.g., represented as *feature diagrams*, as shown for a product line of graph data structures in Figure 1. The root of a feature diagram represents the SPL itself and remaining nodes represent features of that SPL (e.g., feature COLOR represents coloring of Edges). Features can be optional (depicted with an empty dot) or mandatory (depicted with a filled dot). Optional and alternative features introduce variability into an SPL that provides means to create tailor-made software products. For example, in applications for mobile devices alternative implementations for different hardware may exist and an optional feature like DATACOMPRESSION might be provided to reduce network traffic.

2.1 Feature-oriented Programming

FOP treats the features of a software as fundamental units of abstraction and composition. It allows programmers to compose programs based on modularized features. *Feature modules* implement features as increments in functionality [8]. They are kept separate from each other to comply with the principle of *separation of concerns* [15]. To derive a concrete program a user selects the needed features from an SPL. The corresponding feature modules are composed which results in an SPL instance.

FOP can be used as an extension of different programming paradigms. In this paper, we focus on FOP as an extension of *object-oriented programming (OOP)* using classes as implementation units. In this case, a feature is usually implemented by multiple collaborating classes. However, often only a fraction of a class belongs to a feature and the remaining part to other features. Consequently, the classes have to be decomposed with respect to the features of a software in order to generate classes that contain only needed functionality. In Figure 2, we show a graph product line adopted from [35] which we will use as an example throughout the paper. The SPL consists of classes (vertical dashed bars) that are decomposed along features (horizontal bars). Basic implementation is located in module BASEGRAPH which is extended by feature modules COLOR and WEIGHTED that implement coloring

```
1  //Basic implementation based on OOP
2  class Edge {
3    Node* nSource, *nDest;
4  public:
5    Edge():nSource(0), nDest(0) { }
6    bool isSource(Node* n) {
7      return n==nSource;
8    }
9    void setSource(Node* n) {
10     nSource = n;
11   }
12   void setDest(Node* n) {
13     nDest = n;
14   }
15   void print() {
16     printf("egde");
17   }
18 };
```

```
16 //Extensions needed for feature Color
17 refines class Edge {
18   int color;
19 public:
20   Edge():super(), color(0) { }
21   void setColor(int c) { color = c; }
22   void print() {
23     printf("colored ");
24     super::print();
25   }
26 };
```

```
27 //Extensions needed for feature Weighted
28 refines class Edge {
29   int weightSource, weightDest;
30 public:
31   Edge():super(), weightSource(0), weightDest(0) { }
32   void setSourceWeight(int w) {
33     weightSource = w;
34   }
35   void setDestWeight(int w) {
36     weightDest = w;
37   }
38 };
```

Figure 3. FeatureC++ source code of class `Edge` of a Graph product line.

of edges and weights for nodes of a graph. As known from OOP, classes can have members (class `Edge` has members of class `Node`) and can inherit from other classes (class `UEdge` inherits from class `Edge`). Classes `Edge` and `Node` are *refined* to implement functionality needed for features COLOR and WEIGHTED (depicted with filled arrows). Based on this design, different graph data structures can be generated by composing feature modules. For example, we can derive a simple graph that contains only the basic implementation or a colored graph that additionally contains extensions for feature COLOR by simply combining the according modules.

The approach that we present in this paper is implemented with FeatureC++, an extension of the C++ programming language that supports FOP. In Figure 3, we depict the FeatureC++ source code for class `Edge`. The basic implementation (Lines 1–15) provides functionality needed for every graph. It is refined to implement features COLOR and WEIGHTED (Lines 16–38), which is indicated by the keyword `refines`. Refinements can introduce new members (Line 18 and 29) and extend existing methods (Line 22). In method extensions, the refined method can be invoked using the keyword `super` (Line 24). In FeatureC++, refinements of a class are located in different files and are grouped with classes and refinements of the same feature.

2.2 Binding in OOP

In object-oriented languages, objects interact via messages. In compiled object-oriented languages the actual type of a message receiver can be determined statically (*early binding*) or dynamically (*late binding*). These different binding types are important for SPL development and have benefits and drawbacks [1].

Early Binding. In OOP, early binding is used if the type of an object that receives a message is already known before execution (e.g., at link time). In compiled programming languages like C++ it provides possibilities to optimize a program. For example, it avoids indirections for method invocations by including addresses of methods directly in the program code and allows the compiler to inline methods. Early binding, however, also means that code of methods that are never used may be contained in a program and thus increase the binary size.

Late Binding. Late binding is used in OOP if the actual type of a message receiver is not known statically and is determined dynamically. This is usually done at object creation time or at method invocation time. Programming languages, like C++, that support receiver identification at object creation time store the result until method invocation. In C++, this is done by storing a pointer to a *virtual function table*, created by the compiler based on the virtual methods of a class, within the created object [30]. This results in additional memory that is needed to store the type of the object implicitly as pointer to the virtual function table. Nevertheless, it increases flexibility by resolving the method that is called depending on the dynamic type of a message receiver.

2.3 Feature Binding in SPLs

Early and late binding are also important when composing features. Dynamic composition of features requires late binding because the type of an object changes depending on the configuration of an SPL, i.e., depending on the selected features. In contrast, early binding should be preferred in case of static composition to make use of performance optimizations. There are different possible categorizations to describe the time of feature binding in SPLs. Kang et al. distinguish between *product build time* and *product delivery time* [23] and Lee et al. between *static* and *dynamic* feature binding [28]. For this paper a distinction between static and dynamic binding is sufficient:

Static Binding: Static binding means that features are bound when building the binary of a software product. This includes source code transformations or other preprocessing steps as well as compilation and linking of an SPL.

Dynamic Binding: Dynamic binding means binding in a starting or running program and can depend on the dynamic context of that program.

3. Supporting Different Binding Times

There are a number of approaches to implement SPLs that support static binding, dynamic binding, or a combination of both (for details see the related work in Section 5). In contrast to most of these approaches the goal of our work is to provide means for developing SPLs whose features can be composed statically or dynamically from a single code base. That is, a programmer implements features without binding time in mind. At deployment time the programmer decides how features are bound. Boilerplate code for supporting different kinds of bindings is generated automatically.

In prior approaches, an SPL developer had to chose the binding time before implementation and implements the source code according to this decision, e.g., using design patterns, etc. [12]. Our approach implies that all features of an SPL can be bound statically or dynamically. We implemented it as an extension of FeatureC++ and apply code transformation to ordinary C++ code. To support different binding times we use different backends to generate code according to the chosen binding time. The resulting code is compiled to create a statically composed SPL instance or a dynamically composable SPL. Since dynamic composition of features can become a complex task for an application developer we furthermore provide assistance for dynamic SPL instantiation and validation.

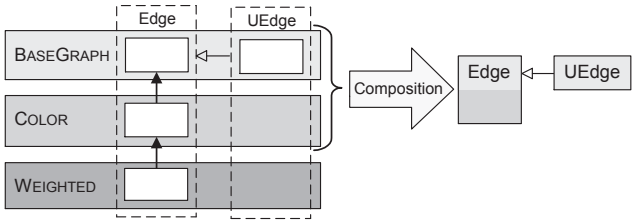


Figure 4. Static composition of classes `Edge` and `UEdge` for a colored graph.

3.1 Static Composition

Using FOP as an extension of OOP, composition of features can be reduced to composition of classes and class refinements. There are different possibilities to implement static composition of classes that are decomposed in feature-oriented approaches, using code generation [8, 26]. FeatureC++ generates for each class with all of its refinements a compound class that consists of:

- the union of all member variables,
- one method for each method refinement,
- one constructor and destructor for each different constructor / destructor definition, and
- one method for each constructor / destructor refinement.

An illustration of the code transformation for class `Edge` (cf. Figure 2) using implementations of modules BASE and COLOR is shown in Figure 4. Using this kind of transformation, the C++ compiler can easily inline method refinements since they are early bound and composed into the same file. Based on such optimizations we could show that FeatureC++ provides the same performance as C++ code that does not support customization [38].

3.2 Dynamic Composition

To support dynamic composition of classes we extended the FeatureC++ code generation process. It transforms the refinement chain of a class into a delegation hierarchy using the *decorator pattern* [17]. This is quite similar to the *Delegation Layers* approach where each layer represents a feature and multiple layers can be combined dynamically [35]. Thus, a class consists of a number of class fragments (*refinements*). Each fragment belongs to a feature and fragments are composed using the decorator pattern. A program is created by composing class fragments of all classes according to the selected features.

In Figure 5 we show the UML representation of code generated for class `Edge` for a colored weighted graph. Each refinement is implemented by a class which is decorated by a class common to all refinements of the target class. For example, class `Edge_Color` represents the refinement of class `Edge` in feature COLOR. The decorator class (`Edge_Decorator`) inherits from an interface `Edge` representing the composed class which can be used in client applications.[2]

[2] C++ does not support interfaces and classes are used instead.

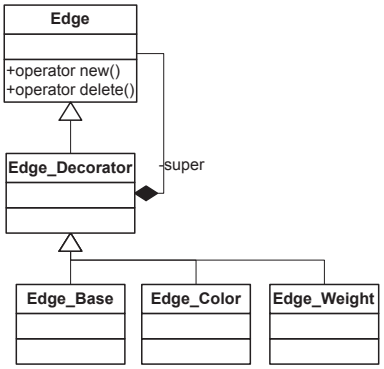


Figure 5. UML diagram of generated code of class Edge of a colored weighted graph.

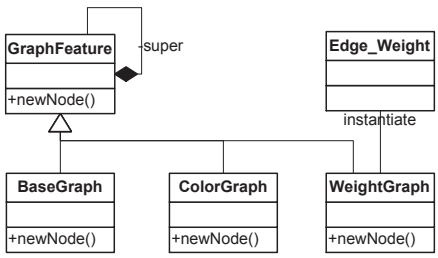
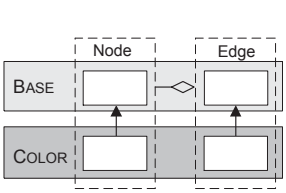


Figure 6. UML diagram of generated code for features of a weighted colored graph.

Feature Classes. When dynamically creating an SPL instance, we have to compose the selected features. We support this *feature instantiation* by using classes to represent features. These *feature classes* are generated in the code transformation process. Much like ordinary classes and refinements, the *feature classes* are also combined using the decorator pattern. The UML representation of the generated code for a part of the graph product line is shown in Figure 6: the classes `BaseGraph`, `ColorGraph`, and `WeightGraph` represent features. They inherit from a generated decorator specific to the SPL (class `GraphFeature`). Each instance of a feature decorator maintains a `super` pointer to the predecessor feature in a composed program.

Class Instantiation. The feature decorators provide factory methods to create instances of ordinary SPL classes (method `newNode` in Fig. 6). Class instances are composed from a number of refinements which means creating an instance for each refinement. Refinement instances are combined by setting up a pointer to the next refinement in the decorator (`super` in Fig. 5). This also enables modifications of the delegation hierarchy at runtime by readjusting the `super`-pointers. The refinement chain thus corresponds to a linked list of class fragments. Changing the configuration of a class corresponds to insertion, exchange, and deletion of elements of this *refinement list*. In each refinement, all calls to `super` are delegated to the next object in the refinement chain.

When a class is instantiated in an SPL (e.g., by calling the `new` operator) the new object has to correspond to the context of creation, i.e., to the SPL instance it is created from. An example for a colored graph and the source code before transformation is shown in Figure 7. An instance of class `Edge` is a composition of base code (Lines 1–7) and its refinement for feature COLOR




```
1  class Edge {
2    Node *nSource, *nDest;
3    void CreateNodes() {
4      nSource = new Node();
5      nDest = new Node();
6    }
7  };
8  //feature Color
9  refines class Edge {...};
```

Figure 7. Class `Edge` that creates new instances of classes `Node` of a colored graph.

```
1  class Edge_Base : public Edge_Decorator {
2    Node *nSource, *nDest;
3    void CreateNodes() {
4      nSource = GetSPLInstance().newNode();
5      nDest = GetSPLInstance().newNode();
6    }
7  };
8  //feature Color
9  class Edge_Color : public Edge_Decorator {...}
```

Figure 8. Class instantiation: code generated for class `Edge`.

(Line 9). In Lines 4 and 5 new nodes are created. Because class `Edge` is part of a colored graph the created nodes also have to be colored. To solve this problem all classes are created by factory methods of the corresponding SPL instance (method `newNode` in Fig. 6). The generated code for class `Edge` is shown in Figure 8. In Lines 4 and 5 instances of class node are created by calling the factory method of the SPL instance. Each implementation of this factory method creates the corresponding refinement instance of class `Node` and appends it to the *refinement list*. For example, method `newNode` in feature class `WeightGraph` creates an instance of `Edge_Weight` and appends it to the refinement instances created by its super classes `BaseGraph` and `ColorGraph`.

Using dynamic composition means that all classes are *virtual classes* [32] of enclosing objects, i.e., the instances of *feature classes*. In FeatureC++ there is no special representation for virtual classes on the source code level (e.g., using nested classes as in other collaboration approaches) and the SPL developer does not have to care about this fact.

Inheritance. When using the decorator pattern for dynamic composition of classes, we also have to support inheritance in an appropriate way. The needed solution also has to provide polymorphism of classes. In FeatureC++, we combine the generated decorators with inheritance of their interfaces to support polymorphic use of classes. The UML representation of the code generated for classes `Edge` and `UEdge` (cf. Section 2.1) is shown in Figure 9. Every class is represented by an interface with the same name (`Edge`, `UEdge`). Inheritance of classes (class `UEdge` inherits from class `Edge`) is represented by inheritance of the corresponding interfaces which supports polymorphic use of that interfaces. In Figure 9, the classes inherit from their interfaces `Edge` and `UEdge`. For a developer of a client application only the interfaces are visible and the concrete implementation is hidden. In client code, classes can be instantiated by using the `new` operators of the interfaces (Figure 9), as we will see below.

To delegate operations of a derived class to its super class the `super` pointer of the first fragment of a derived class refers to the last fragment of the super class. For example, the `super` pointer in `UEdge_Base` in Figure 9 refers to an instance of `Edge_Color`. The corresponding object diagram for an instance of class `UEdge` using

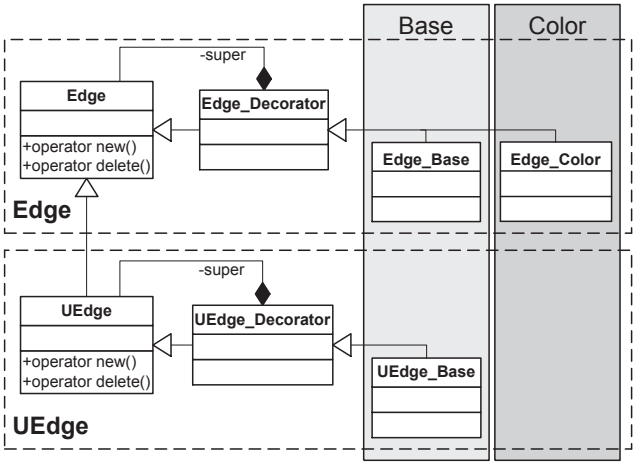


Figure 9. UML diagram of code generated for classes Edge and UEdge of a colored graph.

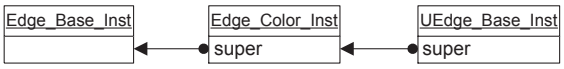


Figure 10. Object diagram of an instance of class UEdge of a colored graph.

feature COLOR is shown in Figure 10. To allow calls to methods implemented in other refinements all references to the *this* pointer in the composed object refer to the actual instance, i.e., the last refinement of a class.

A Two-step Configuration Process. *Overfeaturing* is known from frameworks and means that libraries often contain features that are not needed for a particular client application [13]. That combined with the complex documentation of these frameworks makes the development of client applications highly complicated. In our approach this problem is reduced by selecting only a subset of available features in a first static configuration step. Using FeatureC++, the developer of a client application selects all features intended for dynamic composition as a part of the code generation process before SPL compilation. If it is not known which parts of the SPL are to be used for dynamic composition, all features may be selected. Selecting only a subset of all features decreases the binary size of the resulting application, reduces the interface of the SPL, and thus eases the development of client applications. Furthermore, the FeatureC++ precompiler can generate the documentation of that minimized interface to remove unneeded information.

3.3 Support for C++

We have seen how dynamic and static composition can be supported using different code transformations. Since we use FeatureC++ we can use the transformed C++ code also in client applications written in plain C++.

Static Instances. SPLs may be used as stand-alone applications or as libraries accessed from a client application. Static composition of a FeatureC++ SPL instance results in C++ source code that can be used like traditional object-oriented code. Thus, a client application can create class instances without explicitly creating an SPL instance.

When an SPL itself is used as a stand-alone application the code of a statically created SPL instance has to be executed after application startup. When using C++, a global `main` function is invoked at application startup. In FeatureC++, this `main` function can be defined in any feature outside of a class. It is used by the programmer to create instances of SPL classes and to start the SPL specific execution.

```
1  Edge* connect(GraphSPL& graph, Node* n1, Node* n2) {
2    Edge* e = new(graph) Edge();
3    e->setSource(n1);
4    e->setDest(n2);
5    return e;
6  }

7  int main() {
8    GraphSPL g = BaseGraph();
9    GraphSPL gc = Color(BaseGraph());
10
11   //create and connect nodes of simple graph g
12   Node* nA = new(g) Node();
13   Node* nB = new(g) Node();
14   Edge* e = connect(g, nA, nB);
15
16   //create and connect nodes of colored graph g
17   Node* nColA = new(gc) Node();
18   Node* nColB = new(gc) Node();
19   Edge* eCol = connect(gc, nColA, nColB);
20 }
```

Figure 11. C++ source code of a client application using dynamic composition of different graph product lines and emulation of virtual classes.

Dynamic Instances. Using dynamic composition, an SPL instance is composed from code defined outside the dynamically composable SPL. This also applies to the case where the SPL is used as a stand-alone application because a static entry point for the application is needed. This is done in the same way as described for static composition using a definition of a global `main` method.

If the SPL is used as a library, SPL instances can be created from an independently developed client application. In Figure 11, we show an example for dynamic composition of the graph SPL. A simple graph is created by instantiating the corresponding class `BaseGraph` of the SPL (Line 8) and a colored graph is instantiated by combining a `BaseGraph` with feature `Color` (Line 9). Since the type of a class depends on an SPL instance we mimic virtual classes with C++ by overloading the `new` operator which receives an instance of the SPL that is to be used. In our example, the type of a node depends on the type of an instance of the graph SPL (Lines 12, 13). Nodes can only be instantiated by providing an SPL instance and cause a compile time error otherwise.

The *feature classes* can be used polymorphically as shown for argument `graph` in method `connect` (Figure 11, Line 1). SPL classes can also be instantiated by providing this abstract type instead of a concrete feature class. The correct SPL class instance is created accordingly as shown for class `Edge` in method `connect` (Line 2) where the new instance of `Edge` corresponds to the dynamic type of `graph`. Thus, connecting nodes in Lines 14 and 19 results in different edges for a non-colored and a colored graph.

3.4 An Infrastructure for Dynamic Composition

Static composition of SPLs is often provided by tools that allow programmers to select features and validate a feature selection based on the description of an SPL [6, 37, 9]. This is mostly not the case if dynamic composition is used, even if a feature model is available. Hence, the developer of a client application is responsible for validating the consistency of an SPL instance, which is tedious and error-prone. To ease SPL instantiation and validation of instances we have developed an infrastructure that assists a pro-

```
1  int main(int argc, char** argv) {
2    //load PLM
3    PLM plm;
4    if (!plm.Open(argv[1]))
5      return -1;
6
7    //create instance
8    GraphSPL* graph = plm.CreateInstance(argv[2]);
9    if (!graph)
10     return -1;
11
12   //create nodes of the graph PL
13   Node* n = new(*graph) Node();
14   n->print();
15   return 0;
16 }
17
18 $ graph myGraph.xml Base,Weight,Color
```

Figure 12. C++ source code using a graph product line and automatic instantiation.

grammer with dynamic feature composition. The infrastructure is based on an extensible API.

The SPL-API. The SPL-API is a light-weight programming interface that allows a programmer to dynamically create SPL instances and validate configurations before instantiation. Furthermore, it gives client applications access to a *product line model (PLM)* stored in an XML file [39]. The PLM basically consists of a feature model describing the features of the SPL (cf. Figure 1), domain constraints between features (e.g., mutual exclusion or inclusion of features), and implementation constraints including descriptions of feature interactions and a feature composition order.

Access to the model is provided by the class `PLM` with its main methods `PLM::Open` and `PLM::CreateInstance`. `Open` is used to open a PLM from an XML file and `CreateInstance` to create an SPL instance according to a configuration provided as a list of features. It implicitly validates a given configuration, applies code for feature interactions (as we will describe below), and instantiates the corresponding SPL. The validation process checks a given configuration against constraints of the feature model. These are (1) features not found in the model, (2) missing mandatory features, and (3) invalid feature combinations with respect to feature model constraints. A configuration is invalid if one of the conditions above is not fulfilled. In this case, the instantiation process does not return a new SPL instance but an error. The SPL-API is implemented as a library that is automatically bound to an application via SPL specific code generated in the transformation process. The library is also used to check the validity of an SPL instance in case of static composition.

An example for an instantiation of an SPL is displayed in Figure 12. The application can be started as shown in Line 18 providing the file describing the feature model (`myGraph.xml`) and the features to use (`Base,Weight,Color`). The PLM is opened in Line 4 and an instance is created and validated in Line 8. The resulting instance can be used to create SPL class instances (Line 13). Since the configuration is given in textual form (a list of features), it can be provided at runtime without knowing the needed features of an SPL at client development time.

An Extensible Solution for Dynamic Composition. Manual instantiation of SPLs as shown in Figure 12 can be avoided if the SPL is used as a stand-alone application. We support instantiation of such SPLs by a code generation process at SPL build-time. The generated code is similar to the code presented in Figure 12. It validates a configuration, instantiates an SPL, and executes the main processing method of that SPL. To support arbitrary instantiation

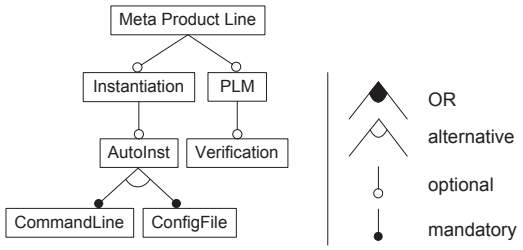


Figure 13. Meta product line to support automatic dynamic product instantiation.

scenarios, we have developed the instantiation and validation functionality as a product line itself. This *meta product line* is developed independently of the *domain SPL*[3] and allows us to generate the appropriate instantiation and validation code tailored to the needs of the application domain.

The feature diagram of the meta product line is presented in Figure 13. Feature AUTOINST encapsulates the functionality needed for automatic SPL instantiation. Currently we support command line arguments (feature COMMANDLINE) or a configuration file (feature CONFIGFILE) to provide an SPL configuration. Validation of SPL configurations is implemented by feature PLM. To automatically instantiate products of a particular domain SPL, the meta product line is extended in the code generation process with code specific to the domain SPL. To support arbitrary scenarios (e.g., loading a configuration from network), classes of the meta product line can be extended manually. These extensions can be part of the meta product line itself, the domain SPL (i.e., domain specific instantiation code), or might be developed as an own product line to support reuse in other SPLs.

Handling Feature Interactions. Crosscutting concerns can be modularized using languages like FeatureC++; however, interactions between crosscutting features also have to be handled. These *feature interactions* often enforce a particular composition order of features and can result in special interaction code. This is also the case for dynamic composition and leads to a complicated SPL instantiation process.

A correct ordering of features is needed to create a semantically correct instance of an SPL. For example, when developing a feature SYNCHRONIZE for the graph product line it has to be ensured that synchronization occurs before accessing members of SPL classes (e.g., accessing weights of nodes). As a result the synchronization feature has to be applied after other features (e.g., feature WEIGHT) to ensure correct synchronization. This ordering of features has to be provided by the SPL developer and is stored as a relative order in our PLM. Using the composition capabilities of the SPL-API the correct ordering of features of a concrete SPL instance is created according to the relative ordering of features at runtime.

If special code is needed to implement the interaction of two or more features this interaction code (a.k.a. *derivatives*) can be modularized and separated from the interacting features [31]. From a technical point of view, a derivative is not different from other features but is only needed if the interacting features are present in an SPL instance. As an example, consider our graph product line using features WEIGHT and SYNCHRONIZE. To avoid concurrent setting of weights we also need to synchronize the access to the weight members stored in edges. Since arbitrary graphs without weights and also without synchronization are allowed we have to modu-

[3] We refer to the developed SPL as *domain SPL* to distinguish it from the meta product line.

larize the interaction between features WEIGHT and SYNCHRONIZE, i.e., the code that synchronizes access to weights. Our PLM includes derivatives as model elements that are related to the interacting features. At SPL build-time, the FeatureC++ precompiler generates binary code for derivatives and the SPL-API applies a derivative in the dynamic composition process if all features that the derivative belongs to are present in an SPL configuration. For example, if we instantiate a synchronized weighted graph we also apply the derivative WEIGHT/SYNCHRONIZE.[4]

4. Discussion and Evaluation

In the following, we will discuss limitations of the presented approach and compare static with dynamic composition with respect to performance and memory consumption.

4.1 Limitations

Our goal is to support static and dynamic composition of SPLs based on a single code base. Supporting C++ as a client language, however, introduces specific problems. Furthermore, semantic differences between statically and dynamically composed code partially limits our approach.

Static Fields and Methods. Static constructs cause problems when using dynamic composition. While it is possible to refine static methods when using static composition this is problematic in case of dynamic composition since the type of the class is dynamic. This raises the question whether a static method or field should actually be the same for all SPL instances (globally static) or only static with respect to one SPL instance. To support the latter option a transformation of static members into members of the SPL feature classes is needed. Unfortunately, it is not possible to access such methods or fields from C++ code by using an SPL instance and the name of the class (e.g., `<spl>.<class>::<method>()`) as known from virtual classes in other programming languages. We currently support static fields and methods only in base modules but think that the correct transformation should use SPL instances as containers for static fields or methods to allow different variants of static methods for different SPL instances. A modification of the FeatureC++ language might be needed to support appropriate access to such methods. To also support globally static code, functions and values defined outside classes might be used but further analysis has to follow to find the most appropriate solution.

Virtual Classes. The emulation of virtual classes for C++ has some deficiencies compared to virtual classes supported directly by the programming language. For example, the interfaces of our virtual classes include all methods required for any dynamically composable feature. This is inherent to the static representation of a dynamically defined class in source code and can be handled by dynamic type checking. In our current implementation an exception is thrown if methods are invoked that are not implemented by the receiver object, i.e., a runtime type error is generated. However, we have not implemented a complete runtime type check. Using plain C++ for compilation of client code does not allow us to statically type check client code which is in general possible [16]. To fully support virtual classes in client code this has to be supported on the language level, e.g., as a future extension of FeatureC++.

C++ Compile Time Constructs. C++ provides constructs that are evaluated at compile time or are part of the static type system and are not available at runtime. Examples are typedefs or enumerations. Such constructs can be used in statically composed SPLs without limitation. When using dynamic composition, however, these constructs cannot be changed or introduced depending on a dynamic configuration since they do not have a representation at runtime. For example, it is not possible to extend an enumeration at runtime which is only checked at compile time and corresponds to an integer value at runtime. For that reason, these constructs can be used with dynamic composition but are available independent of the dynamically selected functionality.

A different problem involves the construct `sizeof` which evaluates the size of an object at compile time. In case of dynamic composition it is not clear how the correct transformation to C++ should look like. There are two possible solutions: the evaluation is still done statically and results in the size of the referenced object in memory, i.e., the size of the interface which is used in the transformed code. The second solution uses a dynamic evaluation and returns the complete size of an object which is composed from several subobjects (the instances of refinements). This dynamic evaluation results in the actual size of memory an object occupies and is what a programmer would expect to be returned. However, this object is not stored in sequence in one memory block but as a list of subobjects. Furthermore, when using `sizeof` to calculate the size of an object to allocate memory for a number of objects of that type the statically evaluated size of the object (i.e., the size of an instance of the interface class) has to be used. Hence, both solutions cause semantic differences between static and dynamic composition.

Further Semantic Issues. There are more semantic differences when comparing static and dynamic composition. One reason are the differing implementations of a class in a statically composed SPL instance and a dynamically composed one. Hence, an SPL developer cannot make any assumptions on the memory layout of objects to avoid semantic errors caused by this difference. This, for example, has to be assured in operations where objects are copied or stored by directly accessing the memory (e.g., when serializing). Another difference caused by the memory layout is the usage of stack and heap memory. When storing an object on the stack it is completely stored on the stack in case of static composition. Using dynamic composition, however, only a small part of the object is stored on the stack and remaining subobjects on the heap which results in differences in memory consumption. Such differences impose restrictions on the source code and require for source code that is written having both approaches in mind. For most scenarios these limitations might be small, but this has to be further analyzed.

Most of the semantic issues presented above are caused by the mixture of high-level and low-level programming constructs that are supported by the C++ language, e.g., that allow direct access to the memory that an object utilizes. Low-level features of the C++ language are useful to support a programming style used to optimize performance or memory consumption, but are inappropriate to abstract from binding time. In future versions of FeatureC++, a new language design has to be applied that removes low-level constructs and abstracts from different binding times.

4.2 Performance and Memory Consumption

For a comparison of static and dynamic composition we will analyze binary size (footprint), consumption of working memory, and performance. In our current implementation, dynamic composition increases the size of an object to store pointers to the predecessor refinement of the refinement chain and a pointer to the last refinement which results in 8 bytes additional memory for each refinement or base class. Furthermore, the object size increases by another 4 bytes to store a pointer to the virtual function table in each refinement. The footprint of dynamically composed SPLs depends on the static pre-selection of features and includes all code that is needed for dynamically composed features. As a result, the loaded binary of a dynamically composable SPL is larger than a statically

[4] We use a listing of all accompanying features to refer to the actual derivative unambiguously. The order of features does not matter.

# of modules	app. footprint [byte]		object size [byte]	
	static	dynamic	static	dynamic
1 (BASE)	2.568	7.600 (6.628)	8	20
2 (+ COLOR)	3.400	9.136 (8.944)	12	36
3 (+ WEIGHT)	3.448	9.984 (9.856)	20	56
4 (+ NAME)	3.648	10.696 (10.696)	24	72

Table 1. Comparison of application footprint and size of instances of class `Edge` for static and dynamic composition. Values in brackets are binary sizes of a statically pre-configured version with dynamic composition.

composed instance as long as dynamic loading of features is not used (e.g., loading features from a dynamic link library).

In Table 1, we show the comparison of the footprint of an application using different variants of the graph SPL and the size of objects for different variants of class `Edge` for static and dynamic composition.[5] The observed binary size of a statically composed application is smaller for all variants compared to dynamic composition. Differences in the footprint of dynamically composed variants are caused by the compiler which can remove unused code that is not referenced by the client application (e.g., the client application does not use method `SetColor` of class `Edge` if the feature is not available). In statically pre-configured dynamic variants, the binary size is further decreased (shown in brackets) but it still results in a larger footprint due to the overhead for dynamic composition.

The size of instances of class `Edge` increases by 12 bytes per refinement in case of dynamic composition. As a result, dynamically composed instances of class `Edge` are in general larger than their statically composed counterparts. However, if the actually needed features are not known at compile time, all features have to be included when using static composition. This would result in a constant object size for static composition of 24 bytes (last variant in Figure 1) and a variable object size of 20–72 bytes for dynamic composition. Hence, edges of a minimal dynamically composed simple graph (only using module BASE) are 17 % smaller than the edges of a statically composed graph which is important in case of a high number of edges. The differences in object size highly depend on the actual size of the refinements. It results in quite worse memory consumption for our example in case of dynamic composition since we used a very small class. This effect decreases with an increasing object size. When considering large refinements and statically not known features, dynamic composition can achieve much better memory consumption than static composition and vice versa. This emphasizes the scenario dependent differences in memory consumption for static and dynamic composition.

Comparing performance, static composition usually does the better job since additional configuration code and indirections for dynamic composition are not executed. This includes an indirection for each refinement if calling a method. Furthermore, statically composed method refinements are inlined by the compiler which results in binary code equal to code that does not support customization. However, dynamic composition can also do better than static composition if the needed configuration is not known at SPL build-time and thus the execution of unneeded code can be avoided. For example, setting colors and weights in edges of our graph degrades performance if these are actually not used. Also in case of dynamic composition performance optimizations are possible and have to be further analyzed.

In summary, the more we know about the needed functionality the better we can optimize an SPL instance, i.e., remove unneeded functionality and choose the correct binding type. Both, memory consumption and performance, thus depend on the application scenario and have to be evaluated in detailed case studies. Nevertheless, it is obvious that a combination of static and dynamic composition on a feature basis bears the potential for optimal performance and memory consumption for different application scenarios.

4.3 Composition Scenarios and Applicability

Dynamic composition has some drawbacks regarding the development effort for client applications when the developer has to handle different SPL instances and has to care about virtual classes and their use. This also has to be considered when deciding which composition technique should be used. However, virtual classes can be hidden from the application developer in some scenarios:

1. **Static composition:** When using static composition the application developer does not have to care about SPL instances or virtual classes at all.
2. **Dynamic composition of a single SPL instance:** When only one dynamically composed SPL instance is needed, the application developer does not have to handle multiple instances and SPL instantiation can be automated and hidden from the developer. An example is a stand-alone application that only depends on configuration data.
3. **Dynamic composition of multiple SPL instances:** If multiple different SPL instances are used the client developer is forced to handle all instances and virtual classes.

Our approach simplifies client development for the first two scenarios. In scenario (2) virtual classes can be avoided, if the client is developed as an SPL. Using FeatureC++, this is possible by creating a client SPL and combining both SPLs at build-time which results in one binary. In this case, the client can access classes of the SPL as if these were classes of the client even though they are dynamically configured.

We implemented the presented approach as an extension of FeatureC++; however, it is a general solution and applicable to other languages, e.g., the feature-oriented Java extension *Jak*.[6] Parts of the implementation are based on C++ language features (e.g., overloading the C++ `new` operator), but these only effect the way client applications have to be written and do not limit the general approach. Nevertheless, we think that client applications should be developed using a language that supports dynamic composition and it became obvious to us that the support for an object-oriented language like C++ can only be a solution to achieve compatibility.

5. Related Work

There are a number of languages and tools that support static as well as dynamic composition. Some of them employ different techniques or paradigms to achieve this. For example, CaesarJ [4] supports static composition based on collaborations and dynamic deployment of aspects. Lee et al. propose to decide before development which features to implement in one component and to compose the resulting components at runtime [27]. In contrast to these approaches, our goal is to decide which composition technique is used not before deployment to enable reuse of source code also with different composition techniques. Chakravarthy et al. provide with Edicts a solution that supports different binding times using different design patterns [12]. The solution, however, is not based on code generation but on manual development of the pattern code.

Other approaches support static as well as dynamic composition based on the same implementation. AspectC++ supports weaving

[5] The introduced feature NAME stores a name for each Edge.

[6] http://www.cs.utexas.edu/~schwartz/

at runtime and at compile time using the same aspects [18]. AspectJ supports weaving advice at compile-time, after compile-time (*post-compile weaving*), and at load-time (when the according class files are loaded into memory) [5]. PROSE [34] and Steamloom [10] furthermore support weaving at runtime and may be combined with AspectJ's static weaving. These AOP approaches can be used to support multiple class extensions at the same time like in collaboration based approaches and FOP. However, there is no direct support for feature composition according to a feature model to support composition validation or avoid invalid instantiation. It is also not possible to easily instantiate SPLs from client code by combining a base product line with class extensions (e.g., defined in an aspect). This has to be done manually by providing aspects to the runtime or load-time weaver which is possible via command-line arguments or configuration files.

Object Teams [19] use dynamic composition of *teams* which represent features. Composition is possible by using statically instantiated *activation teams* which in turn activate other teams [20]. Dynamic composition in Object Teams also considers constraints in the feature model at composition time to avoid invalid SPL instances. Furthermore, it provides an advanced solution for runtime modifications by activating and deactivating teams at runtime. This would also be possible using our approach but is part of further work. Object Teams do not consider interactions between features that result in source code which has to be modularized (i.e., *derivatives*) to support composition of programs that contain arbitrary combinations also of interacting features. This might be possible by implementing a derivative as a team which is applied at product instantiation time. Another difference between Object Teams and FeatureC++ is the correspondence of implemented features to the feature model: while in FeatureC++ a mapping of features to all software artifacts (including implemented roles) defined in folders is used, in Object Teams features are mapped to teams and the source code according to the feature model is generated. Since there is no representation for features in FeatureC++ source code, there are also no modifications of the source code needed if the feature model changes. With Object Teams this results in a redefinition of the mapping from features to roles and collaborations and regeneration of the source code.

Our approach focuses on combining static and dynamic composition. It is based on Delegation Layers [35] which supports dynamic composition of features but currently lacks an implementation. Other collaboration based approaches and layered designs like Jak [7], Java Layers [11], Jiazzi [33], Mixin Layers [40], Aspectual Feature Modules [3], Aspectual Collaborations [29], and Context-oriented Programming [14] also support either static or dynamic composition. In contrast to these approaches, our solution supports both, static and dynamic composition, and assists the developer in dynamically composing SPLs and validation of SPL configurations according to a feature model.

Kegel et al. have shown how inheritance can be automatically refactored into delegation [24]. They showed that both approaches are quite similar and there is no major benefit when using one or the other, but delegation sometimes fails to replace inheritance. We have shown that delegation can replace linear refinement chains which also can be implemented using inheritance as presented by Batory et al. [7]. In our case, delegation does not result in problems when using abstract classes since we do not replace inheritance in general. Instantiation of incomplete class refinements including abstract methods is possible because we generate methods that forward method calls to the next refinement in the refinement chain. If no refinement of a class implements the method an exception is thrown if it is invoked. Currently we cannot statically check if an abstract method is actually implemented by a concrete class instance which could be part of a static type system for FeatureC++.

6. Conclusion and Further Work

Static as well as dynamic composition of SPLs is possible and needed for different application scenarios. We presented an approach to support static and dynamic composition of features from a single code base. This allows us to reuse source code also if the type of composition changes. Furthermore, the decision if static or dynamic composition should be used is postponed until deployment of an SPL. We have implemented the solution as an extension of FeatureC++ and support access to dynamically composable SPLs from client code written in plain C++. When using static composition we provide a code transformation that enables compiler optimizations to achieve high performance.

Another important goal that we wanted to achieve is to assist application developers with composing SPLs dynamically by providing access to a model describing an SPL at a higher level (a product line model). This allows an application developer to validate SPL configurations before composition to avoid invalid programs. By providing support to handle modularized feature interaction code (also called *derivatives*) and hiding it from application developers we can further decrease the complexity of SPL composition at runtime. In the special case of SPLs used as stand-alone applications we furthermore automate SPL instantiation with an extensible approach.

The presented solution is a step toward support for arbitrary binding times on a feature basis, i.e., choosing the binding type for each feature of an SPL separately. This will provide maximal flexibility and opens further possibilities for optimizations tailored to different application scenarios. As a further extension we want to support runtime-adaptable SPLs by loading features after SPL instantiation to further decrease the binary size of an application and to adapt to changes not anticipated at development time.

Acknowledgments

We thank Sagar Sunkle for comments on earlier drafts of this paper. Marko Rosenmüller and Norbert Siegmund are funded by German Research Foundation (DFG), project number SA 465/32-1. Sven Apel's work is funded partly by the German Research Foundation (DFG), project number AP 206/2-1. The presented work is part of the FAME-DBMS project, a cooperation of Universities of Dortmund, Erlangen-Nuremberg, Magdeburg, and Passau funded by DFG.[7]

References

[1] M. Anastasopoules and C. Gacek. Implementing Product Line Variabilities. In *Proceedings of the Symposium on Software Reusability (SSR)*, pages 109–117. ACM Press, 2001.

[2] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer Verlag, 2005.

[3] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008.

[4] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer Verlag, 2006.

[5] AspectJ Team. The AspectJ Programming Guide. Version 1.5.4., Available from http://eclipse.org/aspectj, 2007.

[6] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference*

[7] http://fame-dbms.org/

(SPLC), volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer Verlag, 2005.

[7] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 143–153. IEEE Computer Society Press, 1998.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[9] Big Lever. Gears. http://www.biglever.com.

[10] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 83–92. ACM, 2004.

[11] R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 285–294. IEEE Computer Society, 2001.

[12] V. Chakravarthy, J. Regehr, and E. Eide. Edicts: Implementing Features with Flexible Binding Times. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 108–119. ACM, 2008.

[13] W. Codenie, K. D. Hondt, P. Steyaert, and A. Vercammen. From Custom Applications to Domain-specific Frameworks. *Communications of the ACM (CACM)*, 40(10):70–77, 1997.

[14] P. Costanza, R. Hirschfeld, and W. de Meuter. Efficient Layer Activation for Switching Context-Dependent Behavior. In *Proceedings of the Joint Modular Languages Conference (JMLC)*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103. Springer Verlag, 2006.

[15] E. W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer Verlag, 1982.

[16] E. Ernst. Family Polymorphism. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer Verlag, 2001.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[18] W. Gilani and O. Spinczyk. Dynamic Aspect Weaver Family for Family-based Adaptable Systems. In *Proceedings of Net.ObjectDays*, pages 94–109. Gesellschaft für Informatik, 2005.

[19] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Proceedings of the International Net.ObjectDays Conference*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer Verlag, 2002.

[20] C. Hundt, K. Mehner, C. Pfeiffer, and D. Sokenou. Improving Alignment of Crosscutting Features with Code in Product Line Engineering. *Journal of Object Technology (JOT) – Special Issue: TOOLS EUROPE 2007*, 6(9):417–436, 2007.

[21] F. Hunleth and R. Cytron. Footprint and Feature Management Using Aspect-Oriented Programming Techniques. In *Proceedings of Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES)*, pages 38–45. ACM Press, 2002.

[22] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[23] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented Product Line Engineering. *IEEE Software*, 19(4):58–65, 2002.

[24] H. Kegel and F. Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 431–440. ACM, 2008.

[25] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Verlag, 1997.

[26] M. Kuhlemann, S. Apel, and T. Leich. Streamlining Feature-Oriented Designs. In *Proceedings of ETAPS International Symposium on Software Composition (SC)*, 2007.

[27] J. Lee and K. C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 131–140. IEEE Computer Society Press, 2006.

[28] J. Lee and D. Muthig. Feature-oriented Variability Management in Product Line Engineering. *Communications of the ACM (CACM)*, 49(12):55–59, 2006.

[29] K. J. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual Collaborations – Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, 2003.

[30] S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[31] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. ACM Press, 2006.

[32] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–406. ACM Press, 1989.

[33] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–222. ACM Press, 2001.

[34] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. *SIGOPS Operating Systems Review*, 42(4):233–246, 2008.

[35] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110. Springer Verlag, 2002.

[36] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Verlag, 1997.

[37] pure-systems GmbH. Technical White Paper: Variant Management with pure::variants, 2003–2004. http://www.pure-systems.com.

[38] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems. In *Workshop on Software Engineering for Tailor-made Data Management (SET-MDM)*, 2008.

[39] N. Siegmund, M. Kuhlemann, M. Rosenmüller, C. Kaestner, and G. Saake. Integrated Product Line Model for Semi-Automated Product Derivation Using Non-Functional Properties. In *Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 25–31, 2008.

[40] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.