



Nr.: FIN-013-2009

Combining Static and Dynamic Feature Binding in Software Product Lines

M. Rosenmüller, N. Siegmund, G. Saake, S. Apel

Arbeitsgruppe Datenbanken



Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Technical Report

Nr.: FIN-013-2009

Combining Static and Dynamic Feature Binding in Software Product Lines

M. Rosenmüller, N. Siegmund, G. Saake, S. Apel

Arbeitsgruppe Datenbanken



Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG):

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Marko Rosenmüller
Postfach 4120
39016 Magdeburg
E-Mail: rosenmue@ovgu.de

<http://www.cs.uni-magdeburg.de/Preprints.html>

Auflage: 52

Redaktionsschluss: 10.09.2009

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

Combining Static and Dynamic Feature Binding in 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 (SPL) are used to build similar programs from a single code base. Programs of an SPL can be distinguished in terms of *features*, which represent units of program functionality that satisfy requirements. Features of an SPL can be *bound* either *statically* at program compile time or *dynamically* at runtime. Both binding times have advantages and disadvantages, as we will explain. However, contemporary techniques and tools for implementing SPLs do not allow a programmer to flexibly choose the binding time per feature. We present an approach that integrates static and dynamic feature binding. It allows a programmer to implement an SPL once and to decide later which features are statically bound and which dynamically. We provide a compiler and report from experiences of applying our approach to two non-trivial product lines. We analyze resource consumption of the SPLs and provide a guideline for optimizing resource consumption.

1. INTRODUCTION

Software product line (SPL) engineering has been applied successfully to many domains to generate tailor-made programs.¹ An SPL is a family of similar programs that can be distinguished in terms of *features*. A *feature* is a unit of program functionality that satisfies a requirement, implements a design decision, and provides a potential configuration option [2]. Programs of an SPL are generated by composing modules that implement features. Depending on the underlying composition mechanism, features are either *bound statically* (e.g., at compilation time or in a preprocessing step) or *dynamically* (e.g., when loading a program or at runtime). Both binding times have benefits: static binding facilitates customizability without any cost at runtime whereas dynamic binding allows a programmer to flexibly select and bind features at runtime, however, at the cost of performance and memory consumption [1, 12]. We argue that this tradeoff between static customizability and flexibility due to dynamic binding has to be considered in SPL engineering.

A well known example for static composition are preprocessors (e.g., the C/C++ preprocessor), which enable fine-grained customizability and code optimizations. When composing features statically, however, often only a subset of the features is used at the same time and some features might not be used at all. The reason is that we often cannot decide before deployment or runtime whether a feature is

needed or not. For example, the required functionality of a *database management system (DBMS)*, deployed on a smartphone or a PDA, depends on the requirements of the applications that use the DBMS which may change over time. A Web browser that stores encrypted passwords in a database requires a DBMS with a data encryption feature. This feature is needed only when the Web browser reads or writes passwords, which is typically not the case most of the time. Since power supply, available working memory, and computing power are limited on mobile devices, such a *functional overhead* is not acceptable.

Dynamic binding helps avoiding this overhead by loading features only when they are needed. Additionally, dynamic binding allows others to independently develop and deploy alternative implementations of features or program extensions, e.g., by using plugins. Dynamic binding even provides means for loading functionality on demand from a network. On the other hand, it increases memory consumption and degrades performance especially when many small extensions are used [12]. This *compositional overhead* can be avoided using static binding if the required features are known before deployment. For example, adapters to the underlying operation system do not need to be bound dynamically.

In previous work, we have shown that we can decide after development of an SPL whether static or dynamic binding should be used [27]. However, all features have to be bound either statically or dynamically and it is not possible to use a different binding time for each feature. Hence, we have to choose between customizability at runtime and resource optimizations due to static composition. In this paper, we present an approach that integrates static and dynamic binding seamlessly. In contrast to other approaches, we can choose the binding time of an SPL *per feature* after development and generate *dynamic binding units*, which consist of a user defined set of features. Dynamic binding units are composed at runtime depending on the environment and requirements of the running application. Due to static composition of the *inner* features of a binding unit, we achieve fine-grained customizability and performance optimizations. At the same time, the approach provides high flexibility due to dynamic composition of binding units.

The contributions of this paper are (i) an approach for integrating static and dynamic feature binding in SPLs that allows to flexibly switch the binding time per feature, (ii) an evaluation of the approach regarding customizability and resource consumption, and (iii) a guideline for building binding units to optimize resource consumption of an SPL. Us-

¹http://www.sei.cmu.edu/productlines/plp_hof.html

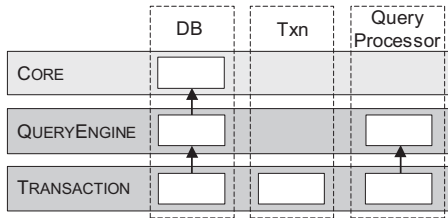


Figure 1: Decomposition of classes (vertical bars) along the features (horizontal bars) in a DBMS.

```

1 //Core implementation
2 class DB {
3     bool Put(Key& key, Value& val) { ... }
4 };

5 //feature QueryEngine
6 refines class DB {
7     QueryProcessor queryProc;
8     bool ProcessQuery(String& query) {
9         return queryProc.Execute(String& query);
10    }
11 };

12 //feature Transaction
13 refines class DB {
14     Txn* BeginTransaction() { ... }
15     bool Put(Key& key, Value& val) {
16         ... //transaction specific code
17         return super::Put(key, val);
18     };
19 };

```

Figure 2: FeatureC++ source code of class DB.

ing code transformations and a generated infrastructure for automating dynamic composition, features can be implemented with the same mechanism independent of their binding time. This simplifies development and reduces implementation effort compared to existing approaches. In our evaluation, we present experiences and insights from applying the approach to two non-trivial product lines in order to analyze the impact of dynamic binding on resource consumption. Based on the results, we analyze which features should be composed into a dynamic binding unit. Moreover, we show how resource consumption of an SPL can be optimized by changing their binding units.

2. FEATURE-ORIENTED PROGRAMMING

In this Section, we introduce feature-oriented programming (FOP), a paradigm for implementing SPLs [24, 8] which we use as the basis for our approach. FOP treats the features of an SPL as fundamental elements of the development process. It allows programmers to implement features as increments in functionality [8]. A user creates a concrete program from an SPL by selecting a set of features that satisfy her requirements. The corresponding *feature modules*, i.e., the implementation units of features, are composed to generate a tailor-made program.

In FOP, a feature module consists of classes and class fragments as shown in Figure 1 for a DBMS product line. The DBMS consists of a CORE implementation and two features QUERYENGINE and TRANSACTION, displayed as vertical bars. Class DB provides the interface of the DBMS and classes Txn and QueryProcessor are used to implement

transactions and query processing. The two features cut across the implementation of multiple classes shown as white boxes. These class *refinements* implement extensions of a class needed for a particular feature. For example, the basic implementation of class DB is provided in the CORE module and extended in features QUERYENGINE and TRANSACTION (depicted with arrows).

We implemented our approach for combining static and dynamic feature binding using *FeatureC++*,² an FOP extension for the C++ programming language [4]. In Figure 2, we depict an excerpt of the FeatureC++ source code of class DB (cf. Fig. 1). Method Put is used to store data provided as key-value pairs. Feature QUERYENGINE adds a new field queryProc and a new method ProcessQuery for processing SQL queries. Feature TRANSACTION adds a new method and *refines* method Put (Line 15). Transaction specific code is added to the beginning of Put (Line 16) and is executed before invoking the refined method using the keyword *super* (Line 17).

3. STATIC AND DYNAMIC FEATURE BINDING

Based on a feature-oriented DBMS implementation as shown in Figure 1, we can generate different DBMS variants by composing a varying set of feature modules. For example, we can derive a simple DBMS only consisting of the CORE implementation or variants that include the features QUERYENGINE and/or TRANSACTION by combining the according modules. The composition of feature modules can either be done statically or dynamically.

Static composition means to combine the code of multiple features into one executable program and dynamic composition means to apply them in a running program or at load-time. There are different possibilities to categorize the binding time of features in SPLs [13]. In this paper, we refer to static binding if a feature is bound in an application before load-time, e.g., at compilation time, and dynamic binding if it is applied at load-time or after loading an application. In prior work on FeatureC++, we have shown that features can be composed statically or dynamically, using the same code base [27]. In the following, we introduce the code transformations used in FeatureC++ to support different binding times. Nevertheless, the presented concepts are language independent and can be applied to other programming languages as well. A more detailed overview of FeatureC++ can be found in [4, 27].

3.1 Static Binding

In order to support static feature binding, the classes of an SPL have to be composed according to the features selected in the configuration process. Since FeatureC++ is based on a source-to-source transformation to C++, the entire code of the base implementation of a FeatureC++ class and their refinements of all selected features is composed into one compound C++ class. This class 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.

²<http://fisd.de/fcc/>

```

1 class DB {
2   bool Put_Core(Key& key, Value& val) { ... }
3
4   Txn* BeginTransaction() { ... }
5
6   bool Put(Key& key, Value& val) {
7     ... //Transaction specific code
8     return Put_Core(key, val);
9   };
10 };

```

Figure 3: Generated C++ source code of class DB using static binding of Core functionality and feature Transaction.

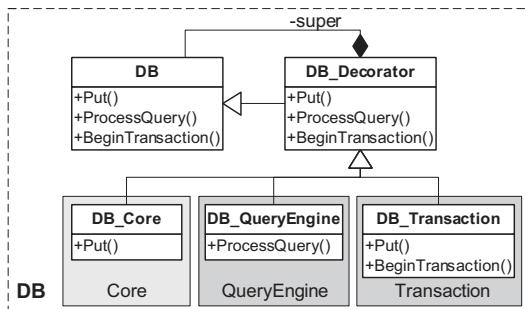


Figure 4: Class diagram of the generated decorator hierarchy for dynamic binding of class DB using features QueryEngine and Transaction.

In Figure 3, we depict the generated C++ code that corresponds to the FeatureC++ code of class DB in Figure 2. This generated code is shown only for illustration and does not have to be read by a programmer that uses FeatureC++. The code corresponds to a composition of the CORE implementation with feature TRANSACTION. All methods and fields except the code of feature QUERYENGINE are composed into one C++ class. The base implementation of method `Put` (feature CORE) was renamed to `Put_Core` (Line 2) to provide a unique name for every transformed method. It is called from its refinement in Line 8. Using this kind of transformation, a C++ compiler can easily inline method refinements since they are composed into the same file. For example, method `Put_Core` is inlined in method `Put` and does not introduce any overhead for method calls. Based on such optimizations, we have shown that FeatureC++ provides the same performance as code that does not provide such fine-grained customizability [25].

3.2 Dynamic Binding

In order to support dynamic binding of features from the same source code, the classes of an application have to be modified dynamically according to the active features. For example, class **DB** (cf. Fig. 2) has to be extended dynamically with code of feature TRANSACTION when activating the transaction management of the DBMS. For that reason, we extended the FeatureC++ code generation process to transform the refinement chain of a class into a delegation hierarchy [27]. Similar to the *Delegation Layers* approach [23], we use the *decorator pattern* [14] to compose classes dynamically. Each class thus consists of a decorator for each refinement and a class is combined dynamically by composing the decorators.

For illustration, we depict the class diagram of the trans-

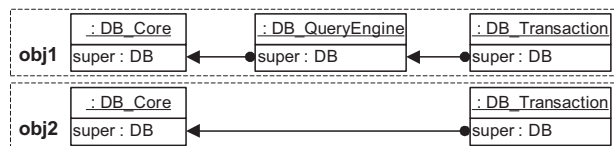


Figure 5: Object diagrams of instances of class DB for two different feature selections.

formed class **DB** in Figure 4. The class is composed from its refinements, which have been transformed into decorators (**DB_Core**, **DB_QueryEngine**, **DB_Transaction**), each belonging to a separate feature. The generated decorator interface (class **DB**) is used to reference dynamically composed classes within the transformed code and also from external source code. The abstract decorator class **DB_Decorator** maintains a reference to the predecessor refinement (`super` reference) and forwards operations that are not implemented by a concrete decorator. The implementation of methods and method refinements are provided by the concrete decorators. For example, method `Put` (Line 3 in Figure 2) and its refinement in feature TRANSACTION (Line 15) are transformed into methods of concrete decorators **DB_Core** and **DB_Transaction** (cf. Fig. 4). Method refinements invoke refined methods by using the `super` reference of the decorator class.

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. For example, when composing feature modules CORE, QUERYENGINE, and TRANSACTION, as shown in Figure 4, there is a feature decorator generated for each feature, which inherits from an abstract decorator, that represents an arbitrary feature of the product line. Each instance of a feature decorator maintains a `super` reference to the predecessor feature in a composed program.

Class Instantiation. Instantiation of dynamically composed classes means to combine objects of the generated concrete decorator classes according to the selected features as depicted in Figure 5. Shown are two different instances of class **DB** using the CORE implementation as well as features QUERYENGINE and TRANSACTION. Each instantiated refinement contains a `super` reference that points to the next refinement in the chain. The dynamically composed objects can be used in the same way as an instance of a regular class and can be modified at runtime by adding or removing instances of decorators. 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*.

For class instantiation, the feature decorators provide factory methods to create instances of ordinary SPL classes which means creating an instance for each decorator. For example, a generated method `newDB()` is used to create an instance for each decorator of class **DB** for a specific SPL instance. Class instances are composed from their decorators within the factory methods of the corresponding features.

