Avoiding Variability of Method Signatures in Software Product Lines: A Case Study

Marko Rosenmüller, Martin Kuhlemann, Norbert Siegmund
School of Computer Science, University of Magdeburg
{rosenmue, mkuhlema, nsiegmun}@ovgu.de

Horst Schirmeier
Department of Computer Science XII, University of Dortmund,
Horst.Schirmeier@udo.edu

ABSTRACT

Software product lines (SPLs) are used to build tailor-made applications based on the features of a domain. Optional and alternative features introduce variability into an SPL that is needed for customization. Variability of method signatures is sometimes necessary to provide additional arguments for optional features. However, this complicates the development of SPLs and is the source of incompatibility between client applications and products of an SPL. In this paper we analyze two techniques to avoid variable method signatures in SPLs and evaluate the solutions in a non-trivial case study.

1. INTRODUCTION

Tailor-made software is needed to fulfill diverse requirements on applications in a domain. *Software product lines (SPL)* are used to build such customizable applications and allow to generate different variants of a software. *Feature-oriented programming (FOP)* [13, 3] and *aspect-oriented programming (AOP)* [9] are paradigms that allow to implement SPLs based on features as extensions to a base program [3].

Client applications communicate with products of an SPL via the SPL interface. This interface has be variable to allow access to the varying functionality of different instances of the SPL. On the other hand, stability of the interface is crucial for communication with other software [12, 4]. This was summarized by Parnas: *"Additional capabilities require adding interface programs but do not require modifications to existing ones"* [11]. Thus when adding new functionality to a software, modifications of the interface have to preserve the existing interface to allow stable communication. This ensures that clients can interact with the SPL independent of additionally configured features.

Features of SPLs can extend existing methods and may also introduce additional arguments. This leads to variability of method signatures if the according feature is optional. Client applications as well as the SPL itself cannot handle this variability since existing method invocations have to be adapted if the signature of a method changes. In this paper we compare two techniques that are promising to reduce negative effects of variability in SPLs by avoiding method signature extensions. We evaluate the techniques in a non-trivial case study and analyze the effects on SPL development and client applications. In the remainder of this paper, we concentrate on FOP, but the presented solutions also apply to other paradigms (e.g., AOP) that are used to develop SPLs.

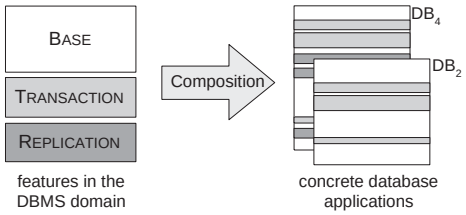

Figure 1: Generating database applications from modularized features.

2. METHODOLOGY

In the following, we give a short introduction to FOP and the importance of interfaces in SPL development. We furthermore present two solutions to avoid variability of method signatures. In our case study and in examples we will use a product line of *database management systems (DBMS)* based on *Berkeley DB*.[1]

2.1 Feature-Oriented Programming

Feature-oriented programming [13, 3] treats *features* of software as basic elements of the development process. Features represent concerns that are of interest to the stakeholder of a software [5] and are the basis for building SPLs.

Feature Modules. *Feature modules* implement features as increments in functionality [3]. These modules are kept separate from each other to comply with the principle of *separation of concerns* [12]. Tailor-made software can be generated by composing selected feature modules.

A simplified example of a feature-oriented design is given in the left part of Figure 1. It shows module BASE and modularized features TRANSACTION and REPLICATION of a DBMS. Basic database functionality is implemented in module BASE and is extended by modules TRANSACTION and REPLICATION that provide transaction processing and replication functionality. Based on this decomposition, different variants of concrete database applications (right part of Figure 1) can be generated by composition of modules.

Decomposition of Classes. Classes are the basic implementation units when using FOP. However, usually only a fraction of a class belongs to one feature and the rest of the class to other features. Figure 2 shows a decomposition

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

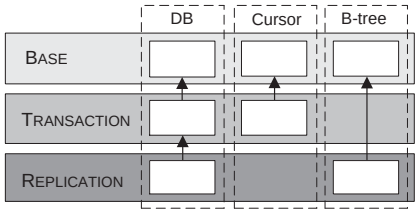

Figure 2: Decomposition of classes (vertical bars) with respect to features (horizontal bars) in feature-oriented designs.

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

```
5  //Extensions needed for feature Transaction
6  refines class DB {
7    TXN* NewTxn() { ... }
8
9    bool Put(Key& key, Value& val) {
10     //Transaction specific code
11     ...
12
13     //Invocation of refined method
14     return super::Put(key,val);
15   };
16 };
```

```
17 //Extensions needed for feature Replication
18 refines class DB {
19   int BeginReplication() { ... }
20 };
```

Figure 3: FOP source code of class `DB` based on the programming language FeatureC++.

of classes `DB`, `Cursor`, and `B-tree` (dashed bars) along the features of Figure 1 (vertical bars). In this design, basic functionality is separated from feature-specific functionality for every class. The classes are refined to implement features TRANSACTION and REPLICATION, which is denoted with an arrow. For example, class `Cursor` is refined to implement the feature TRANSACTION, whereas class `DB` is refined for features TRANSACTION and REPLICATION.

In Figure 3 we show the simplified source code for class `DB` for the design presented in Figure 2. The example is implemented using *FeatureC++*[2], a feature-oriented extension to the C++ programming language [1]. The basic implementation (Lines 1–4) includes functionality needed for every DBMS. Class *refinements* in feature modules TRANSACTION and REPLICATION (Lines 5–20) extend the basic functionality, which is indicated by the keyword `refines`. Such refinements can introduce new members (Lines 7 and 19) and extend existing methods (Lines 9–15). In method extensions, the refined method is invoked using the keyword `super` (Line 14). Classes are composed based on the selection of needed features and include only according functionality. For example, the code in Lines 5–16 of Figure 3 is omitted if the feature TRANSACTION is not used.

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

2.2 Interfaces of Software Product Lines

Interaction with a product of an SPL is possible through the *external interface* of that product [7, 2], i.e., the interface for communicating with the external world. Besides external interfaces, there are also *internal interfaces* within software products. These are used for communication between parts of a software [7, 2] and are only visible internally.

In the context of SPLs we refer to internal and external interfaces as the merged respective interfaces of all features of an SPL. In the scope of this paper, it is sufficient to use the simplified definition of an interface as the merged signatures of classes with their methods. A precise definition of interfaces of software components that includes semantics of usage can be found in [10].

Variability of Interfaces. Access to additional functionality of different variants of products of an SPL is usually provided by extending the external interface. As described for components [12], also modifications of SPLs (e.g., introduced by optional features) have to preserve the existing interface to allow stable communication with other software. Hence, modifications of interfaces should only include addition of classes or class members and removal or modification should be avoided. In SPLs those changes not only occur when a new version of an SPL is released but also if the configuration of a software changes.

Besides changes over time there is another critical effect of variability of external interfaces. When developing an application that has to communicate with a product of an SPL it may be not clear what configuration that SPL instance will have. Other applications, communicating with the same product, may have special requirements that result in a specific configuration. This can lead to incompatibility between clients and the SPL even if there is only one inappropriate extension of the interface (e.g., a modification of a method). As a consequence, the interface of the SPL has to be completely stable except from addition of new elements (e.g., methods).

Variability of internal interfaces causes problems due to SPL-internal method invocations. The SPL thus has to rely on a stable internal interface. Otherwise, dependencies between features arise that prevent, or at least hamper, extension of the SPL.

Passing Arguments. In contrast to object-oriented programming, in AOP and FOP arguments are also passed between method extensions, i.e., between features that extend the same method. Features can add arguments to existing methods and cause a variable method signature if the feature is optional, i.e., a method can have different arguments depending on the used features.

This variability of method signatures causes problems: First, it introduces dependencies to other features that extend methods with a variable signature and thus additionally have to provide the extended signature. Second, classes that are defined within an optional feature, and are thus optional themselves, cannot be used as arguments because they are undefined in the absence of the optional feature.

Figure 4 shows a client application (Lines 8–18) that uses Berkeley DB with transactions (Lines 1–7) to store data.[3]

[3]All presented code samples are based on C++ / FeatureC++ and are slightly modified for simplicity.

```
1  class Database {
2    //create new transaction
3    TXN* NewTxn() { ... }
4
5    //store data
6    bool Put(TXN* txn, Key& k, Value& v) { ... }
7  }
```

```
8  void store_data(Data& d1, Data& d2) {
9    //create new transaction
10   TXN* txn = database.NewTxn();
11
12   //store data
13   if ( database.Put(txn,d1.key,d1.value)
14     && database.Put(txn,d2.key,d2.value))
15     txn->Commit();     //commit transaction
16   else
17     txn->Abort();      //abort transaction
18 }
```

Figure 4: Transactional storage of data in Berkeley DB (Lines 1–7) by a client application (Lines 8–18).

A new transaction is created by invoking `database.NewTxn` (Line 10). Data is stored by calling `database.Put` (Lines 13 and 14) which receives an argument `txn` to provide the transaction to use for storage. If the operations can be finished without an error the used transaction is committed (Line 15) and aborted otherwise (Line 17).

Assuming that feature TRANSACTION is optional, the argument `txn` is only needed if the feature is present in a concrete instance of Berkeley DB. It is thus only available in method `Put` of class `Database` if the feature is configured. Calls to this method have to provide the argument only if it is actually available. This applies for method calls in client applications as well as for method calls inside the SPL. Method signatures have to be stable to avoid such problems.

2.3 Avoiding Variability in Method Signatures

In the following we focus on two solutions to prevent variability of method signatures in SPL development: Forward declarations and a modified SPL design. There are a lot more solutions that provide means to avoid variability of method signatures by using other mechanisms to pass arguments (e.g., variable argument lists). These provide a stable external interface but cannot handle SPL-internal method invocations in an appropriate way: Varying arguments have to be considered in calls to other methods which leads to complex solutions to provide arguments needed for optional features.

Forward Declarations. Arguments introduced in method extensions in optional features lead to variability in the signature of methods. This can be avoided by providing the arguments in all method extensions. If the argument type is a class that is defined in an optional feature it can be declared outside this feature (e.g., in the base application) as an empty class which is known as *forward declaration*. All methods, independent of the enclosing feature, can use the declared but undefined class as an argument in methods. Access to class members is only possible if the according feature, and thus the class itself, is actually defined.

If supported by the programming language, forward declarations can be implemented directly (e.g., in C++) or emulated by defining empty classes to be implemented subsequently in the according feature.

Modified SPL Design. The second approach avoids variability of method signatures by an appropriate design of SPLs. Instead of providing additional arguments in method extensions these are passed to *initialization methods* that store the data within classes of the feature. Methods executed in the following can access this stored data. For example, storing global configuration in configuration classes avoids passing this data as arguments.

This solution comes with two problems: First, in multithreaded applications modifications of those variables have to be synchronized to handle concurrency. Second, data that is only valid for single operations (e.g., a single method call) differs for each thread and thus cannot be simply stored in members of classes.

Storing data within the context of a thread, also known as *Thread Local Storage (TLS)*, overcomes both limitations. It is supported by most C++ compilers by using keyword `__thread` and in Java by using the class `ThreadLocal`. Nevertheless, it can also be implemented in other programming languages. Like static variables, also per thread data should be stored within classes to increase modularity of the source code and to avoid wrong usage. In SPLs this data should furthermore be stored within the according feature to increase feature modularity and to further restrict access.

3. CASE STUDY

In this section we apply the presented solutions to Berkeley DB. We evaluate them with respect to criteria important to SPL development as well as technical criteria.

Berkeley DB is written in the C programming language and contains about 96 thousand lines of code.[4] The developers used C preprocessor statements to allow static configuration. Based on this C version of Berkeley DB we applied a transformation into feature-oriented code (implemented in FeatureC++) and further decomposed the DBMS, including features like transaction management (TRANSACTION) that were not optional before. Our feature-oriented refactoring consists of 36 features with 24 of them being optional.[5]

In the refactoring process of Berkeley DB we recognized 84 interactions between features where 11 are interactions of feature TRANSACTION with other features. Because of this strong interaction and the massive use of additional arguments in method signatures we decided to use feature TRANSACTION for our case study.

3.1 Forward Declarations

The transaction management in Berkeley DB makes use of arguments of type `TXN` in many methods. These arguments provide the transaction to be used for accessing and storing data. When extracting transaction management as a feature we needed to handle these arguments in some way. Introducing a forward declaration of class `TXN` was part of this extraction process and not a separate refactoring. The use of forward declarations allowed us to avoid methods with varying arguments. Thus we could still use the argument after extracting feature TRANSACTION into a separate module.

Figure 5 shows the use of forward declarations in Berkeley

[4]We use the C version 4.4.20 of Berkeley DB.

[5]The source code used in this case study is available under http://wwwiti.cs.uni-magdeburg.de/iti_db/BerkeleyDB/.

```
1  //Forward declaration of class TXN
2  class TXN;
3
4  class Database {
5    //read data
6    bool Get(TXN* txn, Key& k, Value& v) {
7      Cursor* c = Cursor::NewCursor(txn);
8      ...
9    }
10 };
```

```
11 //Feature Transaction
12 class TXN { ... }
13
14 //Refinement of class Database
15 refines class Database {
16   TXN* NewTxn() { ... }
17
18   bool Get(TXN* txn, Key& key, Value& val) {
19     //Transaction specific code
20     TXN* txn = TXN::getCurrent();
21     ...
22     //Invocation of refined method
23     return super::Get(txn,key,val);
24   };
25 };
```

Figure 5: Forward declarations to support arguments of unknown type in Berkeley DB.

DB with the basic implementation (Lines 1–10) and parts of feature Transaction (Lines 11–25). Class `TXN` is declared outside the feature (Line 2) in the base of the SPL and its implementation is postponed (Line 12). The empty declaration of class `TXN` can be used in method signatures within arbitrary features. For example, class `Database` (Lines 4–10) is defined in the base application and method `Get` receives an argument of type `TXN` (Line 6) which is only needed for feature Transaction. The argument is then forwarded to other methods, e.g., to method `NewCursor` (Line 7). Refinements of method `Get` rely on the stable signature including the argument of type `TXN` (Line 18). Instead of providing two variants of the method, with and without argument `TXN`, there is only one variant that includes the transaction argument. Invocations of such methods in client applications have to provide a transaction object or pass `NULL` if transactions are not used.

3.2 Modifying the Design

The extraction of feature Transaction using forward declarations did not change the method signatures in Berkeley DB. Thus we could apply the design oriented approach based on this refactoring. Applying it to Berkeley DB without extracted transactions we would have to use the same refactoring steps again. Hence, it does not affect the result of the case study.

We removed arguments of type `TXN` from all methods in features that are not related to transactions. This required to store transactions on a per thread basis by using TLS (cf. Section 2.3). In case of Berkeley DB, transactions can be nested and we had to use a stack of active transactions per thread that stores transactions in order of creation. In other use cases special solutions might be needed to store the data in an appropriate way.

Figure 6 shows the basic implementation of class `Database` (Lines 1–7) and feature Transaction (Lines 8–30) with

```
1  class Database {
2    //read data
3    bool Get(Key& k, Value& v) {
4      Cursor* c = Cursor::NewCursor();
5      ...
6    }
7  };
```

```
8  //Feature Transaction
9  class TXN {
10 public:
11   TNX()  { curTxn.push(this); }
12   ~TNX() { curTxn.pop(); }
13
14   static TXN* getCurrent() {
15     return curTxn.top();
16   }
17 private:
18   static __thread TxnStack curTxn;
19 };
20
21 //Refinement of class Database
22 refines class Database {
23   bool Get(Key& k, Value& v) {
24     //Transaction specific code
25     TXN* txn = TXN::getCurrent();
26     ...
27     //Invocation of refined method
28     return super::Get(k,v);
29   }
30 };
```

Figure 6: Storing transactions in Berkeley DB in Thread Local Storage implemented in FeatureC++.

class `TXN` and a stack of nested transactions (Line 18). These are protected by using a private declaration (Lines 17–18). In our implementation transactions are stored on creation and removed from storage on destruction (Lines 11–12). The current transaction object can be accessed from methods within feature Transaction (Line 25).

We also removed the argument from methods of the external interface of the SPL. When invoked, these methods automatically use the currently active transaction in the context of the calling thread. Transactions are initialized as usual by calling method `Database::NewTxn`. Thus there is no additional initialization needed.

We observed that access to the currently active transaction (as shown in Line 17 of Figure 6) causes problems (1) if an SPL-internal method should not use the current transaction for special calculations and (2) if a different transaction than the one currently stored has to be used within a method. In the first case, it has to be ensured that no transaction is used what we achieved by setting the current transaction to `NULL` before method invocation. The latter case is similar and the correct transaction has to be stored temporarily. In both cases the last active transaction has to be restored after method execution.

3.3 Evaluation

Comparison of SPL Design. The presented solutions are opposite approaches in the sense of handling additional arguments of optional features. Using forward declarations leads to the union of all arguments in the interface of an SPL. The second approach, storing parameters within their according feature, removes additional arguments that belong to optional features from all methods.

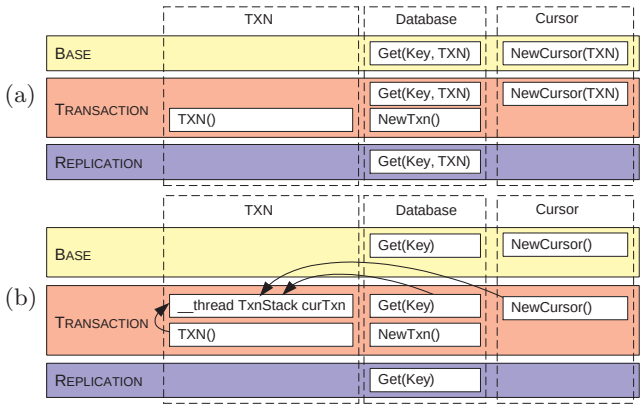

Figure 7: Collaboration of classes `TXN`, `Database`, and `Cursor` using forward declarations (a) and a modified design with TLS (b).

Figure 7 summarizes the design of Berkeley DB when using forward declarations (a) and the TLS approach (b). It shows the collaboration of classes `TXN`, `Database`, and `Cursor` where features are displayed as horizontal bars that cut across the classes (vertical dashed bars). Methods, method extensions, and member variables of classes are displayed within their according features. Access to stored transactions is indicated with arrows.

By using forward declarations, the parameter `TXN` has to be provided in methods `Get` and `NewCursor` as well as in their refinements (Figure 7a). In contrast, the design oriented approach completely modularizes feature TRANSACTION (Figure 7b). Arguments of type `TXN` can be omitted in the signatures of both methods. Transactions are stored on creation of class `TXN` in the context of the calling thread and are accessed only from methods in feature TRANSACTION.

The TLS approach introduces a higher complexity and leads to an increased development effort since a design is needed that allows to store data instead of passing it as arguments. The refactorings that we used can be partially automated but a large amount of the work are design modifications. We argue that the modified design can be of great benefit for development and maintenance since it provides complete feature modularity. A problem that is inherent to the TLS design is a difficult debugging process: When using forward declarations the transactions passed as arguments can be easily inspected and their origin can be determined. This is problematic for the TLS approach.

Memory Consumption. When using forward declarations, arguments of type `TXN` are stored on the stack in method calls. This results in additional memory consumption for each method call. Since there are usually more method invocations, because of argument forwarding to other methods, the totally used memory depends on the call graph. Using the TLS approach, transactions are stored on the heap. Furthermore, there is a stack of transactions needed for each thread that consumes additional memory. For a small number of threads this is negligible.

Comparing both approaches, forward declarations may lead to a higher memory consumption on the stack and TLS approach increases the use of heap memory. Considering the overall memory consumption both approaches do not differ significantly.

Performance. The overhead for passing arguments to another method in C++ is very small. Using TLS to store transactions avoids argument passing but increases CPU utilization due to access to TLS. In the TLS approach further CPU time is used to allocate memory for the stack of transactions (highly depending on the implementation) and to store and read transactions. The estimated overhead needed for the TLS approach is small compared to the time needed to store data in Berkeley DB using transactions ($<$ 1 %) but can be a problem for other use cases. The TLS approach utilizes more CPU time compared to the use of forward declarations.

Impact on Client Applications. Both approaches reduce modifications of client applications through a stable external interface. Forward declarations provide the union of additional arguments of optional features. This leads to complex method signatures which might be confusing for programmers, especially if there are many additional arguments. Clients that invoke such methods have to provide the arguments even if not needed. Another deficit of forward declarations is missing support for the evolution of an SPL: When new features are developed and introduce additional arguments the method signatures have to be extended and may break existing clients.

By removing additional arguments from the external interface in the TLS approach client applications do not have to provide these arguments. This also holds for the evolution of the SPL if new features are added. All available arguments of optional features of an SPL are moved to initialization methods. This is error-prone and leads to incorrect behavior of a feature if it is forgotten. In Berkeley DB we did not need additional initialization methods since initialization of transactions is needed independent of our refactoring.

Summary. Before extracting the transaction management, 189 methods in Berkeley DB used arguments of type `TXN`. After refactoring by using forward declarations 185 methods and 42 method extensions used the argument. Using an appropriate design and TLS we removed `TXN`-arguments from 94 (51 %) methods. The remaining 91 methods (49 %) still use arguments of type `TXN` but are only called from feature TRANSACTION or depending features. 61 (33 %) methods were only forwarding the argument to other methods. We also removed the argument from 15 methods of the external interface of the SPL.

Table 1 compares both approaches. Better results or fulfilled requirements are indicated with a plus and worse results or not fulfilled requirements with a minus. The design-based solution outperforms the use of forward declarations with respect to criteria important for SPL development. Comparing technical criteria forward declarations are superior. These criteria are critical and can render the design based solution unusable for some scenarios.

Both solutions reduce variability of SPL interfaces for different variants of products. Since they also have substantial deficiencies better solutions are desired. However, neither FOP nor AOP provide means to appropriately support variable method signatures in SPLs. Additionally, external variability of method signatures does not allow for stable

	Forward Declaration	TLS
SPL development		
Separation of concerns	−	+
Signature complexity	−	+
Stable signature:		
• in SPL evolution	−	+
• for different configurations	+	+
Reduction of interactions	±	+
Technical criteria		
Performance	+	−
Development effort	+	−
Debugging	+	−
Fault avoidance	−	−

Table 1: Comparison of forward declarations and modified SPL design using TLS.

communication of clients with instances of an SPL.

4. RELATED WORK

There are numerous mechanisms that allow to add arguments to method signatures. Zdun summarizes support for integration of components with varying interfaces [14] and patterns to pass a variable number of arguments [15]. Applied to SPL development the patterns lead to several problems. For some of the patterns provided in [15] this results in an undefined number of arguments (e.g., in *variable argument lists*) and thus a poorly defined interface of a software. Other patterns like *context objects* avoid this problem. However, passing different argument objects (e.g., context objects or variable argument lists) to other methods within an SPL leads to complex solutions since the arguments differ between methods. Using a global argument object for the whole SPL adds undesired complexity to all method calls.

The *adapter pattern* [6] can be used to support integration of components with variable method signatures by wrapping incompatible component interfaces [14]. However, manual implementation of adapters is not possible for an undefined number of products of an SPL. Code generation might be used but does not provide a stable internal interface.

Research on service-oriented architectures also handles variability of services using wrappers and dynamic adaption [8]. However, at the moment there is no approach available that completely automates the generation of wrappers to connect varying services.

5. CONCLUSION AND FUTURE WORK

Variability is naturally found in SPLs. Sometimes variability of method signatures might be used to provide additional arguments for optional features. This kind of variability complicates design and degrades usability of an SPL. In a case study we have shown that forward declarations as well as an appropriate SPL design can avoid variability in signatures of methods. Forward declarations have shown to be inappropriate because of missing separation of concerns, unneeded complexity of method signatures, and missing support for the evolution of an SPL. On the other hand, a design based approach avoids these problems but increases development effort and can degrade performance.

In future work we will analyze combinations with other approaches to avoid the deficiencies of the presented solutions. Furthermore, we investigate in a generative solution to allow variability of the external interface of SPLs.

6. REFERENCES

[1] 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*, 2005.

[2] T. Barros, L. Henrio, and E. Madelaine. Behavioural Models for Hierarchical Components. In *Proceedings of the International SPIN Workshop on Model Checking of Software*. Spinger Verlag, 2005.

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

[4] K. H. Britton, R. A. Parker, and D. L. Parnas. A procedure for Designing Abstract Interfaces for Device Interface Modules. In *Proceedings of the International Conference on Software Engineering*, 1981.

[5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

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

[7] P. Hnětynka. Component Model for Unified Deployment of Distributed Component-based Software. Technical Report TR 2004/4, Charles University, Praha, 2004.

[8] A. Ketfi and N. Belkhatir. Dynamic Interface Adaptability in Service Oriented Software. In *Proceedings of the International Workshop on Component-Oriented Programming*, 2003.

[9] 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*, 1997.

[10] D. Parnas. Information Distribution Aspects of Design Methodology. In *Proceedings of IFIP Congress*, 1971.

[11] D. Parnas. Software Product-Lines: What To Do When Enumeration Wont Work. In *Proceedings of the 1st Workshop on Variability Modelling of Software-intensive Systems*, 2007.

[12] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):264–277, 1979.

[13] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming*, 1997.

[14] U. Zdun. Some Patterns of Component and Language Integration. In *Proceedings of the European Conference on Pattern Languages of Programs*, 2004.

[15] U. Zdun. Patterns of Argument Passing. In *Proceedings of the Nordic Conference of Pattern Language of Programs*, 2005.