FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++

Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake

Department of Computer Science
University of Magdeburg, Germany
email: {apel,leich,rosenmue,saake}@iti.cs.uni-magdeburg.de

Abstract. This paper presents FeatureC++, a novel programming language which supports Feature-Oriented Programming (FOP) for C++. Besides well-known concepts of FOP languages, FeatureC++ supports several novel FOP language features, in particular multiple inheritance and templates for Generic Programming. Furthermore, FeatureC++ solves, as some other FOP languages, the extensibility problem, the constructor problem, and the problem of hidden overloaded methods. A first contribution of this article is to introduce and discuss the language concepts of FeatureC++. A second contribution is the analysis of current drawbacks of FOP languages. Specifically, we outline four key problems and present three approaches to solve them: *Wildcard-Based Mixins*, *Aspectual Mixin Layers*, and *Aspectual Mixins* that adopt concepts of Aspect-Oriented Programming (AOP) in different ways. These approaches are not exclusive to FeatureC++ and can be easily applied to other FOP languages. Furthermore, we introduce our implemented prototype that already supports most of the presented FeatureC++ language concepts, including Aspectual Mixin Layers. Finally, we present a case study to clarify the benefits of FeatureC++ and its AOP extensions.

1 Introduction

Feature-Oriented Programming (FOP) [4] is an appropriate technique to cope with the problems of the software crisis [10]. This is documented by successful case studies, e.g. [3, 1, 5, 6]. Current research on modern programming paradigms such as FOP focuses on Java. *AHEAD* and the *AHEAD Tool Suite (ATS)*[1] are prominent examples [4]. Although used in a large fraction of applications like operating systems, realtime and embedded systems, or databases and middleware C/C++ is rarely considered. Current solutions for C++ utilize templates [27], simple language extensions [25], or C preprocessor directives. These approaches are complicated, hard to understand, and not applicable to larger software systems. This motivated, this article presents FeatureC++[2] a language proposal for FOP in C++. Besides basic concepts, known from other FOP languages,

[1] http://www.cs.utexas.edu/users/schwartz/Hello.html
[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/

FeatureC++ exploits useful concepts of C++, e.g. multiple inheritance or templates, to name a few.

Further contributions of this article are our investigations in the symbiosis of FOP and *Aspect-Oriented Programming (AOP)* [14]. In this regard, our considerations are not restricted to C++. FeatureC++ acts only as a representative FOP language. At first, we discuss some well-known problems of FOP, in particular the lack of crosscutting modularity [22]. We argue that some features of AOP can help to solve these problems. Mainly the ability to implement dynamic crosscutting and the avoidance of method shadowing, as well as the growing acceptance, motivates us to choose AOP. We see several promising approaches for this symbiosis (as we will explain): *Aspectual Mixins*, *Aspectual Mixin Layers*, and *Wildcard-based Refinements*. We discuss the pros and cons of these approaches and the consequences for the programmer.

A further benefit of FeatureC++ is the solution of different problems of object-oriented languages, namely (1) the constructor problem [26, 12], which occurs when minimal extensions have to be unnecessarily initialized, (2) the extensibility problem [13], which is caused by the mixture of class extensions and variations, and (3) hidden overloaded methods in C++, which are hindering for step-wise refinements (as we will explain).

Based on these considerations, we present our prototypical implementation of FeatureC++, which is based on the *PUMA* code transformation system[3]. We explain how to utilize PUMA to implement FeatureC++ and give an overview on already implemented language features. To implement AOP extensions, we utilize *AspectC++* [29], an aspect-oriented language extension to C++. We discuss Aspectual Mixin Layers as our preliminary AOP extension and give an overview of Aspectual Mixins and Wildcard-based refinements as well as consequential implementation issues.

Finally, we introduce a case study and explain how to use FeatureC++. Moreover, we discuss its advantages compared to common FOP approaches by example.

The remaining article is structured as follows: Section 2 gives necessary background information. In Section 3, we introduce the language elements of FeatureC++, and in Section 4 AOP-specific extensions. Afterwards, Section 5 presents our first prototypical implementation and discusses open questions. In Section 6, we discuss a case study. Finally, Section 7 reviews related work and Section 8 concludes the paper.

2 Background

Pioneer work on software modularity was made by Dijkstra [11] and Parnas [24]. Both have proposed the principle of *separation of concerns* that suggests to separate each concern of a software system in a separate modular unit. According to this papers this leads to maintainable, comprehensible software that can easily be reused, configured, and extended.

[3] PUMA: http://ivs.cs.uni-magdeburg.de/~puma/

AOP was introduced by Kiczales et al. [14]. The aim of AOP is to separate crosscutting concerns. Common object-oriented methods fail in this context [14, 9]. The idea behind AOP is to implement so called orthogonal features as *aspects*. This prevents the known phenomena of code tangling and scattering. The core features are implemented as components, as with common design and implementation methods. Using join point specifications, an aspect weaver brings aspects and components together.

FOP studies feature modularity in program families [4]. The idea of FOP is to build software by composing *features*. Features are basic building blocks, which satisfy intuitive user-formulated requirements on the software system. Features refine other features incrementally. This *step-wise refinement* leads to a layered stack of features. Mixin Layers are one appropriate technique to implement features [27]. The basic idea is that features are often implemented by a collaboration of class fragments. A Mixin Layer is a static component encapsulating fragments of several different classes (Mixins) so that all fragments are composed consistently. Mixin Layers are an approved implementation technique for component-based layered designs. Advantages are a high degree of modularity and an easy composition [27]. *AHEAD* is an architectural model for FOP and a basis for large-scale compositional programming [4]. The *AHEAD Tool Suite (ATS)*, including the *Jak* language, provides a tool chain for AHEAD based on Java.[4]

3 FeatureC++ Language Overview

FeatureC++ is a C++ language extension to support FOP. The following paragraphs give an overview of the most important language concepts.

3.1 Introduction to Basic Concepts

To implement FeatureC++, we have adopted the basic concepts of the ATS: Features are implemented by Mixin Layers. A Mixin Layer consists of a set of collaborating Mixins (which implement class fragments). Figure 1 depicts a stack of three Mixin Layers $(1-3)$ in top down order. The Mixin Layers crosscut multiple classes $(A-C)$. The rounded boxes represent the Mixins. Mixins that belong to and constitute together a complete class are called refinement chain. Refinement chains are connected by vertical lines. Mixins that start a refinement chain are called *constants*, all others are called *refinements*. A Mixin A that is refined by Mixin B is called *parent* Mixin or parent class of Mixin B. Consequently, Mixin B is the *child* class or

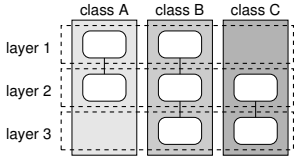

Fig. 1. Example stack of Mixin Layers.

[4] http://www.cs.utexas.edu/users/schwartz/Hello.html

child Mixin of *A*. Similarly, we call Mixin Layers that are refined *parent layers* and the refining layers *child layers*. In FeatureC++ Mixin Layers are represented by directories of the file system. Therefore, they have no programmatic representation. Mixins are represented by included source files. An equation file specifies which features are required for a configuration. Using the feature names the directory search path is browsed to find the corresponding directories (Mixin Layers). Those Mixins found inside the directories are assigned to be members of the enclosing Mixin Layers.

3.2 Syntax of Basic Language Features

To reuse approved language concepts and to increase the users acceptance, FeatureC++ adopts the syntax from the ATS intern *Jak* language [4]. The following paragraphs introduce the most important language concepts:

Constants and Refinements. Each constants and refinement is implemented as a Mixin class inside exactly one source file (*.fcc*-file). Constants form the root of refinement chains (see Fig. 2, Line 1). Refinements refine constants as well as other refinements. They are declared by the *refines*-keyword (Line 5). Usually, they introduce new attributes (Line 6) and methods (Line 7).

```
1  class Buffer {
2    char *buf;
3    void put(char *s) {}
4  };
5  refines class Buffer {
6    int len;
7    int getLength() {}
8    void put(char *c) {
9      if(strlen(c) + len < MAX_LEN)
10       super::put(c); }
11 };
```

Fig. 2. Constants and refinements.

```
1  class Buffer {};
2
3  // two buffer variations
4  class FileBuffer : Buffer {};
5  class SocketBuffer : Buffer {};
6
7  // buffer extension: sync. support
8  refines class Buffer { Lock lock; };
```

Fig. 3. Deriving variations vs. extensions.

Overriding Methods. Refinements can override methods of their parent classes (see Fig. 2, Line 8). To access the overridden method the *super*-keyword is used (Line 10). *Super* refers to the type of the parent Mixin. It has a similar syntax to the Java *super*-keyword, and has similar meaning as the *proceed*-keyword of *AspectJ* and *AspectC++*.

3.3 Advanced Language Features

Solving the Extensibility Problem. FeatureC++ solves the *extensibility problem* [13]: implementation added to a class by creating a new subclass leaves the class' existing subclasses outdated. It is caused by the divergence of variation and extension. Imagine an abstract buffer class with several subclasses, e.g., *FileBuffer*, *SocketBuffer*. These classes are buffer variations. If one wants to extend

(by subclassing) the buffer class using common C++, e.g., by synchronization support, the buffer variations (the other subclasses) are not affected.

FeatureC++ solves the extensibility problem as follows: extensions are expressed as refinements whereas variations are derived using common inheritance. The variations *FileBuffer* and *SocketBuffer*, depicted in Figure 3, inherit from the most specialized form of *Buffer* (in our example the synchronized buffer) regardless of their position and the position of the extension in the refinement chain. This facilitates the easy localized extension of (abstract) classes and the attended automatic extension of all variations.

Constructor Propagation. FeatureC++ solves the constructor problem [26, 12]: in common object-oriented languages, e.g., Java and C++, constructors are not inherited automatically and have to be redefined for each subclass. The idea of FOP is to refine exiting classes by many minimal extensions. In almost all cases these extensions do not need explicit new initializations. FeatureC++ solves the constructor problem by propagating all constructors of parent classes to their subclasses. That means, that all defined constructors of a refinement chain are available in the resulting generated class.

3.4 C++-Specific Language Features

In the considerations so far, we have introduced features that are adopted from Jak. The following language features are novel to FOP and exploit C++ capabilities:

Multiple Inheritance. Multiple inheritance is a powerful concept of object-oriented languages. Figure 4 depicts a buffer refinement that adds synchronization and logging support using multiple inheritance. The corresponding functionality is implemented by inheriting from *Semaphore* and *Logging* and overriding the buffer functions.

Class and Method Templates. Although, FeatureC++ provides Mixin Layers not all generic problem solutions can be expressed sufficiently. Consequently, FeatureC++ supports class and method templates. Figure 5 depicts a buffer refinement, that uses a template parameter to determine the storage data type at instantiation time.

Overloaded Method Propagation. In contrast to Java, standard C++ does not allow to access overloaded methods of a base class. The code snippet depicted in Figure 6 would produce an error: The *put* method of *Buffer* is hidden from external access. In case of object-oriented programming, this is not a big problem, but using FOP this becomes more serious. As mentioned, the key idea of FOP is to successively refine programs by minimal extensions. Following this paradigm, it is often the case that methods are overloaded. Using C++, the programmer has to redefine and delegate calls to overloaded methods explicitly,

```
1 refines Buffer : public
2     Semaphore, Logger {};
```

Fig. 4. Refining a buffer with synchronization and logging support.

```
1 refines template <class T> class Buffer {
2   void push(T &) {}
3   T& pop() {}
4 };
```

Fig. 5. Declaring a refinement as template.

```
1 class Buffer { void put(int i) {} };
2
3 class StringBuffer : public Buffer {
4   void put(char *string) {} };
5
6 int main() {
7   StringBuffer buf;
8   buf.put(4711);
9 }
```

Fig. 6. Accessing overloaded methods from extern is prohibited in C++.

or alternatively to qualify the overloaded method with the *using* keyword. FeatureC++ overcomes this tension by propagating all overloaded methods to the refining layers respectively.[5]

Further Language Features. C++ supports a lot of language features which are not available in Java. Currently, we support refinements of *destructors* and *structs*. Furthermore, we redefine the keyword *this* to additionally provide access to the type of the enclosing Mixin. `this::Buffer` refers to the type of the current position in the refinement chain, instead of the type of the composed class.

Grammar Overview. Table 1 summarizes the grammar of FeatureC++. We understand the rules as extension to the C++ grammar rules.

rules	**description**
layer <layer name>	declares the name of the enclosing Mixin Layer
refines <class declaration>	refines constants or refinements
super::	refers to the type of the parent Mixin
super::<method name>	invokes a method of the parent class
super::<attribute name>	accesses an attribute of the parent class
this::<class name>	refers to the class type of the current layer

Table 1. FeatureC++ grammar overview.

[5] We discuss to introduce a keyword (*propagate*) to enable and disable automatic propagation of inherited overloaded methods.

4 Aspect-Oriented Extensions

FOP has several well-known problems in crosscutting modularity [22]. In this contribution we focus on the problems presented in the following section and discuss the potential benefits of AOP.

4.1 Problems of FOP

No Support for Dynamic Crosscutting: FOP does not support the modular implementation of dynamic crosscuts. A Mixin Layer is a static crosscut that refines multiple implementation units. It is applied at instantiation time and is independent of the control flow at runtime. Feature binding specifications such as "bind feature A to all calls to method m that are in the control flow of method c and only if expression e is true" cannot be expressed. Instead, using the *cflow* and *if* pointcuts, AOP allows to bind aspects in such a way.

Hierarchy-Conform Refinements: Using FOP, feature refinements depend on the structure of their parent features. Usually, a feature refines a set of classes, adds, and overrides methods. Each affected class, method, or attribute must be refined explicitly. In fact, the programmer is forced to construct his features similar to the existing features. This becomes problematic if new features are implemented at a different abstraction level. We clarify this by an example (see Sec. 6). As basic feature we consider a stock information broker. This feature should be refined by a pricing feature. Whereas the broker is expressed in terms of stock information, requests, brokers, clients, and database connections, the pricing feature is expressed using the intuitive producer-consumer-pattern. FOP is not able to change the abstraction level accordingly [22]. Instead, AOP is able to implement non-hierarchy-conform refinements by using wildcards in pointcut expressions.

Excessive Method Shadowing: The problem of excessive method shadowing occurs when a feature crosscuts a large fraction of existing implementation units. For instance, if a feature wants to add multi-threading support, it has to override lots of methods to add synchronization code. Mostly this code redundant, e.g. setting lock variables. AOP deals with this problem by using wildcards in pointcut expressions to specify a set of target methods (join points).

Interface Extensions: The problem of interface extension frequently occurs in incremental designs. Often a programmer adds only a delta to an existing feature by overriding existing methods. Some of such extensions demand for an extended interface of the overridden methods. For instance, a client-server application that allows to send messages between client and server shall be refined by a session protocol. Intuitively, a programmer refines the client' send method to implement the session protocol. In some situations, the programmer has to pass a session id to the send method. This leads to an extended interface of the overriding methods. Indeed, using some workaround

this problem can be solved. But AOP with its pointcut mechanism is much more elegant.

4.2 Combining FOP and AOP Concepts

In the following, we present our investigations in solving the above discussed problems and present three approaches that adopt AOP language concepts.

Wildcard-Based Refinements. Our first idea, to prevent a programmer from excessive methods shadowing and hierarchy-conform refinements were *Wildcard-Based Refinements*. The key idea is to refine a whole set of parent Mixins instead of refining one Mixin by another one only. Such sets are specified by wildcards. Figure 7 shows two Mixins that use wildcards to specify the Mixins and Methods they refine. The unspecified sub-strings are denoted by '%'. The first Mixin refines all classes that start with "*Buffer*" (Line 1). The second refines all methods of Buffer that start with "*put*" (Line 3). The meaning of the first type of refinement is straightforward: The term *Buffer%* has the same effect as one creates a set of new refinements for each found Mixin that matches the pattern (*Buffer%*).

```
1 refines class Buffer% {};
2
3 refines class Buffer {
4   void put%(...) {} };
```

Fig. 7. Two Wildcard-Based Mixins.

Using wildcards in method refinements yields some difficulties. If a programmer uses only wildcards to match methods, with a fixed signature, e.g. *void put%(char *c)*, the refining method can access these arguments and call the parent methods by *super*. In case of an unspecified argument list, e.g. *void put%(...)*, the refining method does not know the arguments of the parent methods. In this case a reflective API can help. Lohmann et al. show how to preserve static type checking in reflectively accessing arguments [17].

Wildcard-based Mixins are similar to static introductions of AspectJ and AspectC++. However, they can be seamlessly integrated into Mixin Layers and support the FOP paradigm.

Aspectual Mixin Layers. The key idea behind *Aspectual Mixin Layers* is to embed aspects into Mixin Layers. Each Mixin Layer contains a set of Mixins and a set of aspects. Doing so, Mixins implement static and hierarchy-conform crosscutting, whereas aspects express dynamic and non-hierarchy-conform crosscutting. In other words, Mixins refine other Mixins and depend, therefore, on the structure of the parent layer. These refinements follow the static structure of the parent features. Aspects refine a set of parent Mixins by intercepting method calls and executions as well as attribute accesses. Therefore, aspects are able to implement dynamic crosscutting and non-hierarchy-conform refinements.

Figure 8 shows a stack of Mixin Layers that implements some buffer functionality, in particular, a basic buffer with iterator, a separated allocator, synchronization, and logging support. Whereas the first three features are implemented

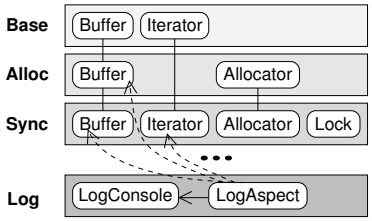

Fig. 8. Implementing a logging feature using Aspectual Mixin Layers.

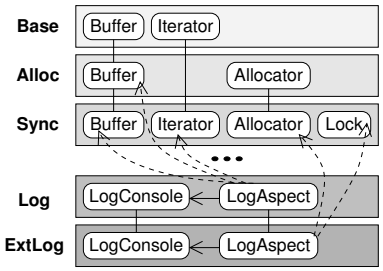

Fig. 9. Refining an Aspectual Mixin Layer.

as common Mixin Layers, the *Logging* feature is implemented as an Aspectual Mixin Layer. The rationale behind this, is that the logging aspect captures a whole set of methods that will be refined (dashed arrows). This refinement is not hierarchy-conform and depends on the runtime control flow (dynamic crosscutting). Moreover, the use of wildcard expressions prevents the programmer of excessive method shadowing. Without, Aspectual Mixin Layers the programmer has to override all target methods explicitly.

A further highlight of Aspectual Mixin Layers is that, aspects can refine other aspects by using the *refines* keyword. To access the methods and attributes of the parent aspect, the refining aspect uses the *super*-keyword. Figure 9 shows an Aspectual Mixin Layer that refines the logging aspect by additional join points to enhance the set of refined methods. Beside this, the logging console (implemented as a Mixin) is refined by additional functionality, e.g. a different or extended output format.

Aspects can not only refine the methods of parent aspects, but also of pointcuts. This allows to easily reuse and extend of existing join point specifications (as in the logging example).

To express aspects in Aspectual Mixin Layers we adopt the syntax of AspectC++. Figure 10 depicts an aspect refinement that extends a logging feature, including a logging aspect. It overrides a parent method in order to adjust the output format (Line 2) and refines a parent pointcut to extend the set of target join points (Line 3). Both is done using the *super*-keyword.

```
1 refines aspect LogAspect {
2   void print() { changeFormat(); super::print();  }
3   pointcut log() = call("%␣Buffer::put(...)") || super::log();
4 };
```

Fig. 10. Aspect embedded into a Mixin Layer.

Aspectual Mixins. The idea of *Aspectual Mixins* is to apply AOP language concepts directly to Mixins. In this approach, Mixins refine other Mixins as with common FeatureC++ but also define pointcuts and advices (see Fig. 11). In other words Aspectual Mixin are similar to Aspectual Mixin Layers but integrate

pointcuts and advices directly into its Mixin definition. In the following, we discuss some important differences:

```
1 refines class Buffer {
2   int length() {}
3   pointcut log() = call("% Buffer::%(...)");
4 };
```

Fig. 11. Combining Mixins and AOP elements.

The subset of the Mixin, which implement AOP elements, is called *aspectual subset* of the *overall* Mixin. This combination reveals some interesting issues: Using Aspectual Mixins the instantiation of aspects is triggered by the overall Mixin instances. Regarding the above presented example, the buffer Mixin and its aspectual part are instantiated as many times as the buffer. This corresponds to the *perObject* qualifier of AspectJ. However, in many cases only one aspect instance is needed. To overcome this problem, we think of introducing a *perObject* and *perClass* qualifier to distinguish these cases. This, however, introduces a second problem: If an aspect, part of an Aspectual Mixin, uses non-static members of the overall Mixin it depends on the Mixin instance. In this case, it is forbidden to use the *perClass* qualifier. FeatureC++ must guarantee that *class-bound* Aspectual Mixins, especially their aspectual subset, only access static members of the overall Mixin instance. In case of *instance-bound* Aspectual Mixins this is not necessary.

4.3 Discussion

All three approaches provide solutions for some problems of FOP. Whereas Wildcard-Based Mixins only solve the problem of hierarchy-conform refinements, method shadowing, and interface extensions (due to reflective access), the Aspectual Mixin and Aspectual Mixin Layer can solve all stated problems. However, the Aspectual Mixin approach yields some problems regarding the instantiation. Moreover, it is currently not clear if the mixture of aspectual and Mixin subsets leads to deeper problems. At the current state, Aspectual Mixin Layers are the only implemented variant (see Sec.5).

A further highlight of all three AOP extensions is a specific bounding mechanisms that supports a better incremental design. Originally it was proposed by Lopez-Herrejon and Batory [19]. They argue that with regard to program family evolution, features should only affect features of former development stages. Current AOP languages, e.g. AspectJ and AspectC++, do not follow this principle. This decreases aspect reuse and complicates incremental design. In contrast to common AOP languages, all discussed extensions follow this principle. See Section 5 for a detailed discussion of this bounding mechanism used in Aspectual Mixin Layers.

Finally, we want to emphasize that all three approaches are not specific to FeatureC++. All concepts can be applied to Jak/AHEAD and AspectJ, as well as similar languages.

5 Prototype Implementation

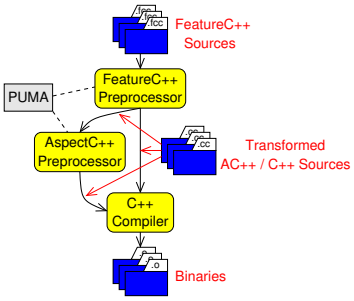

Fig. 12. Overview of FeatureC++ implementation.

Extending C++ to support FOP a modified syntax is needed. Since the difference of C++ and FeatureC++ is relatively small, it is appropriate to utilize a source-to-source transformation. FeatureC++ is implemented as a C++ preprocessor and uses PUMA for code transformation. The input FeatureC++ code is transformed into native C++ code in case of mixins and into AspectC++ code in case of Aspectual Mixin Layers. Figure 12 depicts the overall architecture. This section introduces PUMA and how PUMA is used to implement FeatureC++.

5.1 PUMA

PUMA (PURE Manipulator) is a library that provides functions for scanning and parsing C++ code [28]. Based on the resulting *abstract syntax tree (AST)* one can analyze and manipulate the code structure in a high-level way. The modified AST can be saved to disk as header and source files. Currently, PUMA is used to implement *AspectC++*, an aspect-oriented language extension of C++ [29].

5.2 Using PUMA to Extend C++

To support FOP in C++, we have to extend the syntax of C++. At first, we have modified the scanner to introduce new keywords, e.g. *refines*, *super*, etc. Furthermore, we have extended the grammar of C++ by creating a new grammar file and generating a parser using *Lemon*[6]. Doing so, PUMA is enabled to parse FeatureC++ sources and build a FeatureC++ AST. The set of source files is specified by an input equation file[7]. It contains the names of the features as well as their desired arrangement. The order of the features is important to infer the right arrangement of the refinement chains. To transform

[6] http://www.hwaci.com/sw/lemon/lemon.html

[7] The term "equation file" is adopted from AHEAD. It defines ordered feature collections as algebraic expressions.

FeatureC++ sources into C++ sources the AST is restructured. A proprietary transformation unit, which is the core of FeatureC++, analyzes the FeatureC++ AST, looks for FeatureC++-specific keywords, and substitutes them by C++-specific counter parts. Note that these transformations are not trivial because the FeatureC++ keywords cannot be mapped to C++ keywords one-to-one. In fact, the transformation often depends on global knowledge of the FeatureC++ code structure and affects a lot of code positions, e.g. to introduce includes or forward declarations. After the transformation step the restructured AST is saved to disk as headers and source files. As with Jak [4], it is possible to transform each Mixin of a refinement chain to a separate C++ class or to merge all Mixins into one composed class. Whereas the first approach supports debugging and maintainability, the latter generates small and fast code. In every case, the resulting native C++ code can be compiled by every standard conform C++ compiler. In case of AOP extensions, the aspect code is transformed and passed to the AspectC++ compiler, which generates C++ code as well (see Sec. 5.3).

Example. To further clarify the transformation process, we explain it by our buffer example (see Fig. 13). Scanning this code snippet the FeatureC++ compiler detects the tokens depicted in Figure 14.

```
1 class Buffer {};
2 refines class Buffer
3     : Semaphore {};
```

Fig. 13. Basic and Synchronization feature.

```
1 Layer Base: "class", "Buffer", "{", "}", ";"
2 Layer Sync: "refines", "class", "Buffer", ":",
3             "Semaphore", "{", "}", ";"
```

Fig. 14. Token lists of the Base and Sync features.

Out of the scanned token lists and the FeatureC++ grammar specification FeatureC++ creates two ASTs, each for a layer. Figure 15 shows the AST of the buffer base and the corresponding token list. Since the base buffer class includes no FeatureC++-specific language features the AST is C++ conform. That does not mean that it will not be transformed. At least the name will

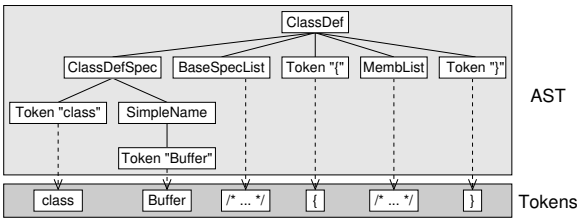

Fig. 15. AST of a simple Buffer class (Base feature).

be changed (*Buffer* → *Buffer_Base*). This is necessary because all member of a refinement chain have the same name. In C++ this leads to an error. Therefore, we change the names depending on their enclosing layers: *Mixin* → *Mixin_Layer*. Figure 16 depicts the AST of the synchronization feature (*Sync*). One can see

that the AST contains FeatureC++ specific nodes, e.g. node *RefinesDef* and token *refines*. In the transformation step the ASTs are restructured as follows:

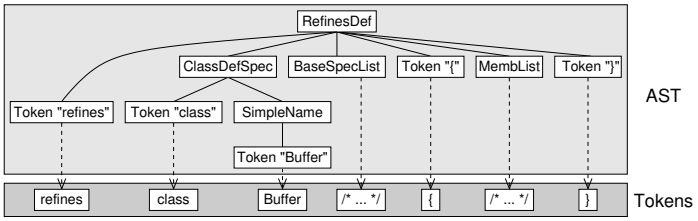

Fig. 16. AST of a Buffer refinement (Sync feature).

The restructuring starts from the base layer and follows the order of the layer stack. In the AST of the base layer only the name is substituted (*Buffer* → *Buffer_Base*). Additionally, in the AST of a refinement the *refines* keyword is substituted by an inheritance declaration (*refines Buffer* → *class Buffer_Sync : public Buffer_Base*). Obviously, the knowledge of the base layers is necessary to substitute the class names. In case of multiple inheritance the other base classes are applied to the resulting list of parent classes (`BaseSpecList`). Figure 17 depicts the transformed C++ conform AST of the Sync-refinement. One can see

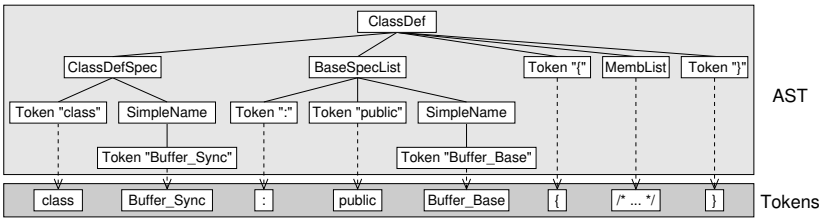

Fig. 17. Transformed AST of a Buffer refinement (Sync).

that the resulting token list follows the C++ standard.

After this transformation step the ASTs are written to C++ source files. All generated classes are embedded into a namespace. The namespace gets the name of the configuration, specified by the file name of the input equation file, e.g. *BufferConf.equation* creates the namespace *BufferConf*. The use of namespaces allows to use different configurations out of the same layer repository, e.g., *BufferConf1::Buffer*, *BufferConf2::Buffer*, etc.

5.3 Some Implementation Issues

In the following, we discuss some special issues of the FeatureC++ implementation.

Extensibility Problem. FeatureC++ solves the extensibility problem as follows: extensions are expressed as refinements and variations using inheritance. The implementation is straight forward: Names of declared types are transformed, by substituting them with a combination of type name and layer name,

but instantiations of types remain unchanged. Recall the buffer example depicted in Figure 3. Figure 18 depicts the transformed C++ code.[8] One can see that

```
1 class Buffer_Base {};
2 class FileBuffer : Buffer {};
3 class SocketBuffer : Buffer {};
4 class Buffer : public Buffer_Base {};
```

Fig. 18. Transformed C++ code of buffer variations and extensions.

```
1 Buffer::Buffer(char *buf) :
2   Buffer_Base(buf) {};
3 void Buffer::put(int i) {
4   Buffer_Base::put(i); }
```

Fig. 19. Refinement with generated *put* method.

the variants of the buffer inherit not from the buffer basis but from the most specialized buffer (the extended buffer).

Method and Constructor Propagation. Implementing method and constructor propagation is relatively trivial. FeatureC++ scans all Mixins (their ASTs) for constructors and overloaded methods. All found constructors and all overloaded methods are applied to the corresponding child mixins. This is done by the transformation of the corresponding ASTs. The generated constructors and methods call the counterparts of the parent classes. Figure 19 shows the generated code of a buffer refinement. The method *put* is generated automatically.

Aspectual Mixin Layers. To implement Aspectual Mixin Layers we utilize AspectC++ [29]. The key feature of Aspectual Mixin Layers is to define aspects inside a Mixin Layer. Mainly, these aspects are handled by AspectC++. AspectC++ gets the aspects in form of source files as input and transforms the already generated C++ code (generated out of the common Mixin) using the aspect specifications accordingly. However, before aspects are passed to AspectC++ some modifications take place:

1. names of aspects are substituted (similar to Mixins)
2. class names with explicit namespace identifier are perceived as external classes, e.g. *std::string* is not substituted
3. *refines*- and *super*-keywords are substituted
4. pointcut expressions are transformed, to match only the classes of the underlying (parent) layers and the current layer

(1) The names are substituted to indicate that the aspects belong to Mixin Layers and to avoid name conflicts. (2) Class names with explicit namespace identifiers are interpreted as external types. Therefore no name substitution is needed. (3) All occurrences of *refines* and *super* are translated similar to common Mixins. In case of refined pointcuts the *super*-keyword is substituted by the refined parent pointcut.[9] (4) To avoid the problem of poincuts that refer to features of future development stages [19] aspects only affect classes of the

[8] Note the depicted classes have to be ordered before compiling (class *Buffer* have to declared at the second position).

[9] Note in common AspectC++ child pointcuts shadow parent pointcuts.

parent layers. To achieve this bounding mechanism the user declared pointcuts must be restructured: Type names inside pointcuts are translated to match only the types of the current and the parent layers. Each pointcut which contains a type name is translated into a set of new pointcuts that refer to all type names of the parent classes. Imagine the synchronization aspect, depicted in Figure 20, is part of a Mixin Layer *Sync* that has two parent layers (*Base*, *Log*) and several child layers. FeatureC++ transforms the aspect and the pointcut as depicted in Figure 21.

```
1 aspect SyncAspect {
2   pointcut sync() :
3     call("% Buffer::add(...)");
4 }
```

Fig. 20. A simple pointcut expression.

```
1 aspect SyncAspect_Sync {
2   pointcut sync() :
3     call("% Buffer_Sync::add(...)")
4     || call("% Buffer_Log::add(...)")
5     || call("% Buffer_Base::add(...);
6 }
```

Fig. 21. Transformed pointcut.

6 A Case Study

This section presents a case study to clarify the use of FeatureC++. In particular, it gives insight in how to implement Aspectual Mixin Layers. We choose the stock information broker example, adopted from [22], in order to point to the benefits of Aspectual Mixin Layers compared to common FOP approaches. In particular, we show how FeatureC++ overcomes the problems discussed in Section 4.

Stock Information Broker. A stock information broker provides information about the stock market. The central abstraction is the *StockInformationBroker (SIB)* that allows to lookup for information of a set of stocks (see Fig. 22). A *Client* can pass a *StockInfoRequest (SIR)* to the *SIB* by calling the method *collectInfo*. The *SIR* contains the names of all requested stocks. Using the *SIR*, the *SIB* queries the *DBBroker* in order to retrieve the requested information. Then, the *SIB* returns a *StockInfo (SI)* object, which contains the stock quotes, to the client.

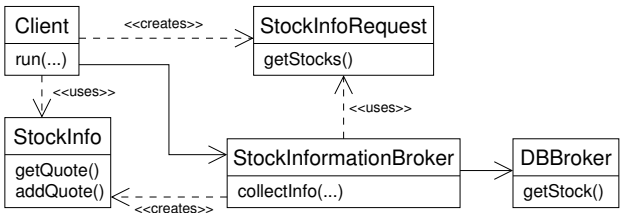

Fig. 22. Stock Information Broker.

All classes are implemented as classes (constants) and are encapsulated by a Mixin Layer (see Fig. 23). In other words, this Mixin Layer implements a basic stock information broker feature (*BasicSIB*).

```
1  class StockInformationBroker {
2    DBBroker m_db;
3  public:
4    StockInfo &collectInfo(StockInfoRequest &req) {
5      string *stocks = req.getStocks();
6      StockInfo *info = new StockInfo();
7      for (unsigned int i = 0; i < req.num(); i++)
8        info->addQuote(stocks[i], m_db.get(stocks[i]));
9      return *info; }
10 };
11
12 class Client {
13   StockInformationBroker &m_broker;
14 public:
15   void run(string *stocks, unsigned int num) {
16     StockInfo &info = m_broker.collectInfo(StockInfoRequest(stocks, num));
17     ... }
18 };
```

Fig. 23. The basic stock information broker (BasicSIB).

Pricing Feature as Mixin Layer. Now, we want to add a *Pricing* feature that charges the clients account depending on the received stock quotes. Figure 24 depicts this feature implemented using common FOP concepts. *Client* is refined by an account management (Lines 16-23), *SIR* is refined by a price calculation (Lines 2-5), and *SIB* charges the clients account when passing information to the client (Lines 10-12).

There are several problems to this approach: (1) The *Pricing* features is expressed in terms of the structure of the *BasicSIB* feature. This problem is caused because FOP can only express hierarchy-conform refinements. It would be better to describe the *Pricing* feature using abstractions as product and customer. (2) The interface of *collectInfo* was extended. Therefore, the *Client* must override the method *run* in order to pass a reference of itself to the *SIB*. This is an inelegant workaround and increases the complexity. (3) The charging procedure of the clients cannot be altered depending on the runtime control flow. Moreover, it is assigned to the *SIB* which is clearly not responsible for this function. (4) An hypothetical accounting functionality that traces and logs the transfers (not depicted) suffers from excessive method shadowing because all affected methods, e.g. *collectInfo*, *price*, *balance*, etc., have to be shadowed.

Pricing Feature as Aspectual Mixin Layer. Figure 25 depicts the pricing feature implemented by an Aspectual Mixin Layer. The key difference is the *Charging* aspect. It serves as an observer of calls to the method *collectInfo*. Every call to this method is intercepted and the client is charged depending on its request. This solves the problem of the extended interface because the client is charged by the aspect instead by the SIB. An alternative is to pass the client'

```
1  refines class StockInfoRequest {
2    float basicPrice();
3    float calculateTax();
4  public:
5    float price();
6  };
7
8  refines class StockInformationBroker {
9  public:
10   StockInfo &collectInfo(Client &c, StockInfoRequest &req) {
11     c.charge(req);
12     return super::collectInfo(req); }
13 };
14
15 refines class Client {
16   float m_balance;
17 public:
18   float balance();
19   void charge(StockInfoRequest &req);
20   void run(string *stocks, unsigned int num) {
21     StockInfo &info = super::m_broker.collectInfo(*this,
22       StockInfoRequest(stocks, num));
23     ... }
24 };
```

Fig. 24. The pricing feature using FOP (Pricing).

reference to the extended *collectInfo* method (not depicted). In both cases, the *Client* does not need to override the *run* method.

A further advantage is that the charging of client' accounts can be made dependent to the control flow (using the *cflow* pointcut). This makes it possible to implement the charging function variable. In this context, the method shadowing is prevented by using wildcard expressions in pointcuts, e.g., for capturing calls to all methods which are relevant for price transfer (accounting feature for tracing and logging transfers). Finally, our example shows that using Aspectual Mixin Layers we were able to refine only these classes that play the roles of product (*SIR*) and customer (*Client*).

Summary. Although the stock information broker example is very simple, it reveals the benefits of FeatureC++ and Aspectual Mixin Layers. FeatureC++ has all advantages of common FOP approaches. Furthermore, it is able to handle dynamic crosscutting, interface extensions, non-hierarchy-conform refinements, and excessive method shadowing. Table 2 summarizes these advantages.

7 Related Work

Work in several areas is related to this contribution: programming support for FOP, solutions for specific problems of object-oriented languages, AOP-related techniques, as well as the combination of AOP and FOP.

Programming support for FOP. One appropriate way to implement features of program families in a modular way are Mixin Layers [27]. Mixin Layers can be implemented using C++ templates [27], *P++* [25], *Jak* [4], and *Java Layers* [7]. The *Jiazzi* component model [21] and the *Delegation Layers* [23] are

```
1  aspect Charging {
2    pointcut collect(Client &c, StockInfoRequest &req) =
3      call("% StockInformationBroker::collectInfo(StockInfoRequest &)")
4      && args(req) && that(c);
5
6    advice collect(c, req) : after(Client &c, StockInfoRequest &req)  {
7      c.charge(req); }
8  };
9
10 refines class StockInfoRequest {
11   float basicPrice();
12   float calculateTax();
13 public:
14   float price();
15 };
16
17 refines class Client {
18   float m_balance;
19 public:
20   float balance();
21   void charge(StockInfoRequest &req);
22 };
```

Fig. 25. The pricing feature using Aspectual Mixin Layers (Pricing).

problem	**solution**	**example**
interface extensions	method interception, argument passing by aspects	the pricing aspect passes the clients reference to the SIB
hierarchy-conformity	refine only structure relevant Mixins; other are modified by aspects	refines *Client* as customer and *SIR* as product
dynamic crosscutting	use specific pointcuts (*cflow*, etc.)	charge clients depending on its runtime state
method shadowing	wildcards in pointcut expressions	match all methods with price transfer

Table 2. Advantages of FeatureC++ Aspectual Mixin Layers.

related and can express Mixin Layers. Jiazzi components can be composed as binaries. Delegation Layers are composable at runtime, but in current it lacks adequate language support.

Solutions for specific problems of object-oriented designs. The constructor problem was first mentioned by Smaragdakis et al. [26]. Java Layers solve it by automatic constructor propagation from parent to child classes [7]. Similarly, Eisenecker et al. utilize static C++ meta-programming [12].

Several approaches solve the extensibility problem, introduced by Findler et al. [13]: Java Layers [7], ATS Mixin Layers [4], Jiazzi [21], Delegation Layers [23].

Aspect-Related Techniques. *AspectJ* [15] and *AspectC++* [29] are prominent aspect-oriented language extensions to Java and C++. They focus on feature modularity using aspects and static weaving. Mezini et al. [22] and Lopez-Herrejon et al. [18, 19] discuss the drawbacks of current aspect bounding mechanisms, in particular, no module boundaries, no feature cohesion, etc. FeatureC++ overcomes most of these tensions.

Hyper/J supports multi-dimensional separation of concerns for Java [30]. This approach to software development is more general than that of FeatureC++ because it addresses the evolution of all software artifacts, including documentation, test cases, and design, as well as code. Hyper/J focuses on the adaptation, integration, and on-demand remodularization of Java code. However, Hyper/J has several similarities to AHEAD [2]. As FeatureC++ can be embedded into AHEAD, it is an appropriate alternative to Hyper/J.

FOG is a C++ language extension to define one class several times in order to compose them compile-time [31]. These different definitions can be interpreted Mixins that implement together one target class.

Combining AOP and FOP. Several approaches aim to combine AOP and FOP. Mezini et al. show that using AOP as well as FOP standalone lacks crosscutting modularity [22]. They propose *CaesarJ* for Java as a combined approach. Similar to FeatureC++ CaesarJ supports dynamic crosscutting using pointcuts. In contrast to FeatureC++ CaesarJ focuses on aspect reuse and on-demand remodularization. *Aspectual Collaborations*, introduced by Lieberherr et al. [16] encapsulate aspects into modules with expected and provided interfaces. The main focus is similar to CaesarJ. Colyer et al. propose the *principle of dependency alignment*: a set of guidelines for structuring features in modules and aspects with regard to program families [8]. They distinguish between orthogonal and weak-orthogonal features/concerns. Loughran et al. support the evolution of program families with *Framed Aspects* [20]. They combine the advantages of frames and AOP in order to serve unanticipated requirements.

8 Conclusion

This paper has presented FeatureC++, a novel language for FOP in C++. FeatureC++ supports language concepts similar to the most FOP languages. Moreover, it supports novel concepts, e.g. multiple inheritance and generic programming support, known from common object-oriented languages. After a short description of problems of FOP languages in implementing program families we have proposed three ways to solve them: *Wildcard-Based Mixins*, *Aspectual Mixin Layers*, and *Aspectual Mixins*. All these approaches are independent of FeatureC++. They combine AOP and FOP features to enhance the crosscutting modularity of FeatureC++. In particular, we support pointcuts, advices, and wildcards. After a discussion of their pros and cons we have presented an overview of our prototypical implementation. Most of the presented language concepts are already implemented. Currently, Aspectual Mixin Layers are the only implemented AOP extension. One can download a preliminary version of FeatureC++ at our web site[10]. Using the case study of a stock information broker we have shown that Aspectual Mixin Layers increase the crosscutting modularity significantly.

[10] http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/

References

1. D. Batory et al. Creating Reference Architectures: An Example from Avionics. In *Symposium on Software Reusability*, 1995.
2. D. Batory, J. Liu, and J.N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. *ACM SIGSOFT*, 2003.
3. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology*, 1(4), 1992.
4. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6), 2004.
5. D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems*, 9(2), 1997.
6. R. Cardone et al. Using Mixins to Build Flexible Widgets. In *Proc. of AOSD*, 2002.
7. R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *Proc. of ICSE*, 2001.
8. A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical report, Computing Department, Lancaster University, 2004.
9. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
10. E. W. Dijkstra. The Humble Programmer. *CACM*, 15(10), 1972.
11. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
12. U. W. Eisenecker, F. Blinn, and K. Czarnecki. A Solution to the Constructor-Problem of Mixin-Based Programming in C++. In *GCSE'2000 Workshop on C++ Template Programming*, 2000.
13. R. Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proc. of the 3rd Int. Conf. on Functional Programming*, 1998.
14. G. Kiczales et al. Aspect-Oriented Programming. In *Proc. of ECOOP*, 1997.
15. R. Laddad. *AspectJ in Action – Practical Aspect-Oriented Programming*. Manning Publication Co., 2003.
16. K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal (Special issue on AOP)*, 46(5), 2003.
17. D. Lohmann, G. Blaschke, and O. Spinczyk. Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In *Proc. of GPCE*, 2004.
18. R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. of ECOOP*, 2005.
19. R. E. Lopez-Herrejon and D. Batory. Improving Incremental Development in AspectJ by Bounding Quantification. In *Software Engineering Properties and Languages for Aspect Technologies*, 2005.
20. N. Loughran et al. Supporting Product Line Evolution with Framed Aspects. In *Workshop on Aspects, Components and Patterns for Infrastructure Software (held with AOSD)*, 2004.
21. S. McDirmid and W. Hsieh. Aspect-Oriented Programming in Jiazzi. In *Proc. of AOSD*, 2003.
22. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT*, 2004.
23. K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proc. of ECOOP*, 2002.

24. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions On Software Engineering*, SE-5(2), 1979.
25. V. Singhal and D. Batory. P++: A Language for Large-Scale Reusable Software Components. In *Workshop on Software Reuse*, 1993.
26. Y. Smaragdakis and D. Batory. Mixin-Based Programming in C++. In *Proc. of GCSE*, 2000.
27. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering Methodology*, 11(2), 2002.
28. O. Spinczyk. Aspektorientierung und Programmfamilien im Betriebssystembau. *Lecture Notes in Informatics (LNI) – Dissertations*, 2003.
29. O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proc. of Conf. on Technology of Object-Oriented Languages and Systems*, 2002.
30. P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. of ICSE*, 1999.
31. E. D. Willink and V. B. Muchnick. An Object-Oriented Preprocessor Fit for C++. *IEEE Proc. on Software*, 147(2), 2000.