FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming

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 language extension to C++ that supports Feature-Oriented Programming (FOP) and Aspect-Oriented Programming (AOP). Besides well-known concepts of FOP languages, FeatureC++ contributes several novel FOP language features, in particular multiple inheritance and templates for generic programming. Furthermore, FeatureC++ solves several problems regarding incremental software development by adopting AOP concepts. Starting our considerations on solving these problems, we give a summary of drawbacks and weaknesses of current FOP languages in expressing incremental refinements. Specifically, we outline five key problems and present three approaches to solve them: *Multi Mixins*, *Aspectual Mixin Layers*, and *Aspectual Mixins* that adopt AOP concepts in different ways. We use FeatureC++ as a representative FOP language to explain these three approaches. Finally, we present a case study to clarify the benefits of FeatureC++ and its AOP extensions.

1 Introduction

Feature-Oriented Programming (FOP) [5] is an appropriate technique to implement program families and incremental designs [7, 3, 1, 6]. It aims to cope with the increasing complexity, lacking reusability and customizability of nowadays software systems. Aspect-Oriented Programming (AOP) [16] is a related programming paradigm and has similar goals: It focuses mainly on separating and encapsulating crosscutting concerns to increase maintainability, understandability, and customizability [9, 17]. However, it does not focus explicitly on incremental designs or program families.

One contribution of this article are our investigations in the symbiosis of FOP and AOP. Our aim is to combine the strengths of both approaches with regard to the implementation program families. Doing so, we firstly review well-known problems of FOP, in particular shortcomings in crosscutting modularity. We argue that certain features of AOP can help to solve these problems. Mainly, the ability to handle dynamic crosscutting and homogeneous crosscuts, as well as the growing acceptance, motivates us to choose AOP. We propose three ways

to do this symbiosis (as we will explain): *Multi Mixins*, *Aspectual Mixin Layers*, and *Aspectual Mixins*. These three approaches cope with the present problems of the FOP paradigm in different ways. They contribute several ideas in improving crosscutting modularity in the face of incremental software development and program families.

Current research in this direction focuses mainly on Java. *AspectJ*[1] and the *AHEAD Tool Suite (ATS)*[2] are prominent examples. Although used in a large fraction of applications like operating systems, realtime embedded systems, or databases C++ is rarely considered. Current solutions for C++ utilize templates [31], simple language extensions [29], or C preprocessor directives. These approaches are complicated, hard to understand, and not applicable to larger software systems. Thus motivated, this article presents FeatureC++[3], a language proposal for FOP in C++. Using FeatureC++, we explain the use and the benefits of the three AOP extensions integrated into a FOP language.

Besides basic concepts known from other FOP languages FeatureC++ further exploits useful concepts of C++, e.g. multiple inheritance or generic programming support. Moreover, it solves different problems of object-oriented languages in implementing incremental designs, namely (1) the constructor problem [30, 13], which occurs when minimal extensions have to be unnecessarily initialized, (2) the extensibility problem [14], 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. Whereas these solutions are known from previous work, the consistent embedding into a C++-based FOP/AOP language is new. We perceive them as indispensable for successful FOP languages.

To underpin our language proposal we have implemented a first prototype, available at our web site. Using a case study, we illustrate how to use FeatureC++. The study reveals the advantages of FeatureC++ and its AOP extensions compared to common FOP approaches.

The remaining article is structured as follows: Section 2 gives necessary background information. In Section 3, we introduce the basic language concepts and features of FeatureC++. Section 4 reviews drawbacks and weaknesses of FOP and suggests three approaches to overcome them. In Section 5, we present a case study that explains the use of FeatureC++ and its advantages. Section 6 reviews related work. Finally, Section 7 concludes the paper.

2 Background

Pioneer work on software modularity was made by Dijkstra [12] and Parnas [27]. Both proposed the principle of *separation of concerns* that suggests to separate each concern of a software system in a separate modular unit. Following this principle leads to maintainable, comprehensible software that can be easily reused, customized and extended.

[1] http://eclipse.org/aspectj/
[2] http://www.cs.utexas.edu/users/schwartz/Hello.html
[3] http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/

AOP was introduced by Kiczales et al. [16]. The aim of AOP is to separate crosscutting concerns. Common object-oriented methods fail in this context [16, 11]. The idea behind AOP is to implement 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 *pointcuts* and *advices*, an aspect weaver brings aspects and components together. Pointcuts specify the *join points* of aspects and components, whereas advices define which code is applied to these points. AspectJ and *AspectC++*[4] are prominent AOP extensions to Java and C++.

FOP studies feature modularity in program families [5]. The idea of FOP is to build software (individual programs) by composing *features*. Features are basic building blocks that 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 [31]. 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. Advantages are a high degree of modularity and an easy composition [31].

AHEAD is an architectural model for FOP and a basis for large-scale compositional programming [5]. *AHEAD* generalizes the concept of features and feature refinements. Features consist not only of code but of several types of artifacts, e.g., makefiles, UML-diagrams, documentation. The *AHEAD Tool Suite (ATS)* provides a tool chain for AHEAD and FOP based on Java. The included *Jak* language supports Java-based Mixin Layers.

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

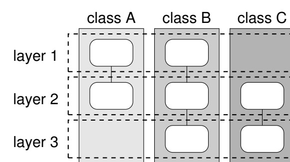

Fig. 1. Stack of Mixin Layers.

[4] http://www.aspectc.org/

that is refined by Mixin B is called *parent* Mixin or parent class of Mixin B. Consequently, Mixin B is the *child* class or child Mixin of A. Similarly, we speak of parent and child Mixin Layers. In FeatureC++ Mixin Layers are represented by file system directories. Therefore, they have no programmatic representation. Those Mixins found inside the directories are assigned to be members of the enclosing Mixin Layers.

3.2 Basic Language Features

To reuse established language concepts and to increase the users acceptance, FeatureC++ adopts the syntax from the *Jak* language. The following paragraphs introduce the most important language concepts and features:

Constants and Refinements. Each constant and refinement is implemented as a Mixin inside exactly one source file. The root of a refinement chain is formed by these constants (see Fig. 2, Line 1). Refinements are applied to constants as

```
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 *s) {
9      if(strlen(s) + len < MAX_LEN)
10       super::put(s); }
11 };
```

Fig. 2. Constants and refinements.

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

Fig. 3. Deriving variations vs. extensions.

well as to other refinements. They are declared by the *refines* keyword (Line 5). Usually, they introduce new attributes (Line 6) and methods (Line 7) or extend[5] methods of their parent classes (see Fig. 2, Line 8). To access the extended 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 a similar meaning to the *proceed* keyword of *AspectJ* and *AspectC++*.

Solving the Extensibility Problem. FeatureC++ solves the *extensibility problem* [14]: implementation added to a class by creating a new subclass leaves the class' existing subclasses outdated.[6] It is caused by the divergence of variation and extension. Imagine an abstract buffer class with several subclasses, e.g., *FileBuffer*, *SockBuffer*. These classes are buffer variations. With common object-oriented languages the extensibility problem occurs: If one wants to extend the

[5] We do not use the term 'override' because we want to emphasize that usually method refinements reuse the parent method. This is more an extension than an overriding.

[6] The original definition regards the extension of designs by new operations and data types. However, following Cardone et al. [7] we deal with this problem in the context of class hierarchy extensions.

buffer class by subclassing, the existent 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 *SockBuffer*, 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 [30, 13]: 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 existing classes by many minimal extensions. In many cases these extensions do not need explicit new initializations. FeatureC++ solves the constructor problem by propagating all constructors of parent classes 'down' to their subclasses. That means, that all defined constructors of a refinement chain are available in the resulting generated class.

Besides constructors, also hidden overloaded methods are propagated down the refinement chain. Background is that C++ does not allow to access overloaded methods of a base class. These *hidden* methods are propagated too (see [2] for more details).

3.3 C++-Specific Language Features

The section so far has introduced features that are mostly adopted from Jak. The following language features are novel to FOP and exploit C++ capabilities. This makes FeatureC++ more powerful than current approaches, e.g. in supporting generic programming.

Multiple Inheritance. Multiple inheritance is a useful concept of object-oriented languages to express refinements. 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 extending the buffer functionality.

```
1 refines Buffer : public
2     Semaphore,
3     Logging {/*...*/};
```

Fig. 4. Refining a buffer using multiple inheritance.

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

Fig. 5. Declaring a refinement as template.

Generic Programming. To implement generic solutions, FeatureC++ supports generic programming, in particular class and method templates. Generic

programming is essential to program families. The ability to parameterize refinements improves the variability in composing individually customized programs.

Figure 5 depicts a buffer refinement that uses a template parameter to determine the storage data type at instantiation time. Method templates are used analogously.

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 overload the keyword *this* to additionally providing 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.

4 Aspect-Oriented Extensions

FOP has several well-known problems in modularizing crosscutting concerns [24]. These problems degrade the modularity of program family members and decrease maintainability, evolvability, and customizability. We investigate solutions for the following selected problems, which are relevant for program family development: (1) weaknesses in expressing dynamic crosscutting, (2) inability to express homogeneous crosscutting concerns, (3) refinements have to be hierarchy-conform, (4) problem of method interface extensions, and (5) excessive method extensions.

We briefly review these problems (see [2, 24] for a further discussion).

1. FOP has weaknesses in expressing dynamic crosscutting, which e.g. depends on the runtime control flow. FOP copes mainly with static crosscutting. Dynamic crosscutting is only supported in terms of intercepting and extending methods. AOP languages handle dynamic crosscutting in a more elegant and robust way. Novel innovative pointcut approaches (e.g. [26]) show the strength of AOP in this respect.
2. A second problem is that FOP languages deal only with heterogeneous crosscutting concerns, which apply different code at different positions. AOP, instead, copes mainly with homogeneous concerns that extend the base code at different join points with the same code fragments.
3. A third problem is that refinements to a given feature base must match the structure of this base, in particular, the class structure. A reorganization of the structure or the raising to a new abstraction level, as described in [24], is not possible.
4. A further problem occurs if method refinements need to extend the signature of the refined method, i.e. the argument list. This is only possible with an inelegant workaround.
5. A final problem are excessive method extensions in case of refinements that crosscut a large fraction of existing classes. For each method a crosscut depends on, the programmer has to introduce an extended method. This problem is caused by the inability of FOP to modularize homogeneous crosscutting concerns.

We perceive solutions to the listed problems as a benefit for implementing incremental designs and as an improvement of FeatureC++ against common FOP approaches. In the following, we present our investigations in solving these problems using AOP language features as wildcards, pointcuts and advices. We present only preliminary approaches. A detailed analysis of the impact of these approaches on real-world applications, robustness, and code quality is part of future work.

Multi Mixins. Our first attempt was to tackle the problems of excessive method extensions and hierarchy-conformity. The idea is to allow Mixins to refine a whole set of parent Mixins instead of refining only one parent Mixin. Because of this refinement multiplicity we call these Mixins *Multi Mixins*.

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

Fig. 6. Two Multi Mixins.

The sets of parent Mixins are specified by wildcards. Figure 6 shows two Multi 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 semantics of such *Class Multi Mixins* are straightforward: The term *Buffer%* has the same effect as if one creates a set of new refinements for each found Mixin that matches the pattern (*Buffer%*). The second Multi Mixin, called *Method Multi Mixin*, refines all methods of Buffer that start with "*put*" (Line 3). Similar to AOP languages, a join point API provides access to the arguments.

Both types of Multi Mixins ease the encapsulation of static homogeneous crosscuts by using wildcards to specify the set of target join points. Furthermore, Multi Mixins solve the problem of excessive method extensions by refining multiple methods using one extension. In this way also the hierarchical structure of the parent Mixin Layer is changed.

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, heterogeneous, and hierarchy-conform crosscutting, whereas aspects express dynamic, homogeneous, 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 advanced dynamic crosscutting and homogeneous, non-hierarchy-conform refinements.

Figure 7 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 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

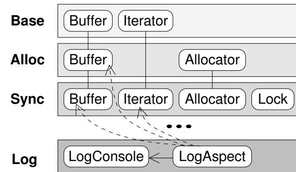

Fig. 7. Implementing a logging feature using Aspectual Mixin Layers.

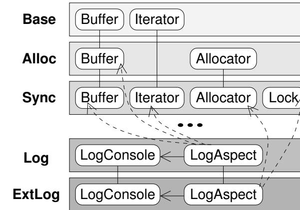

Fig. 8. Refining an Aspectual Mixin Layer.

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

Fig. 9. An aspect embedded into a Mixin Layer.

not hierarchy-conform and depends on the runtime control flow (dynamic crosscutting). Moreover, the use of wildcards prevents the programmer from excessive method extensions.

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 8 shows an Aspectual Mixin Layer that refines the logging aspect by additional join points to extend the set of intercepted methods. Beside this, the logging console (implemented as a Mixin) is refined by additional functionality, e.g. a modified output format. Generally, aspects can refine the methods of parents aspect as well as the parent pointcuts. Extending pointcuts increases the reuse of existing join point specifications (as in the logging example). Note that refining/extending aspects is conceptually different than applying aspects themselves. Whereas the latter case applies the aspects first, the former case results in a transformation of the aspect code before applying them to the target program.

To express aspects in Aspectual Mixin Layers we adopt the syntax of AspectC++. Figure 9 depicts an aspect refinement that extends a logging feature, including a logging aspect. It extends 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.

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 they also define pointcuts and advices (see Fig. 10). In other words, Aspectual Mixins are similar to Aspectual Mixin Layers but integrate pointcuts and advices directly into Mixins.

In this sense, Aspectual Mixins are related to *Classpects* [28] that unify AOP and OOP language concepts. However, we see problems regarding this intermixing of Mixins (attributes, methods, refinements) and aspect elements (pointcuts, advices). Combining both may lead to a dependency of life time of the aspect and the Mixin subset. Usually, aspects are not instantiated directly by the user

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

Fig. 10. Combining Mixins and AOP elements.

but triggered by the matching join points (as in AspectJ). Instead, Mixins are instantiated by the user. Currently, we are not sure what the instantiation of an Aspectual Mixin results in: (1) The aspect subset is instantiated as well (as in *Caesar* [24]). (2) The aspect subset is instantiated only once (as in *AspectJ*). The problem of the former case is that often only one instance is needed. The latter case may lead to problems in accessing the internals of the Aspectual Mixin. For instance advices must not access instance attributes of the enclosing Aspectual Mixin. A deeper analysis of the consequences is important and part of future work.

4.1 Summary

All three approaches provide solutions for certain problems of FOP. They deal with the problems in different ways and contribute improved techniques for implementing incremental designs. Whereas Multi Mixins only solve the problem of hierarchy-conform refinements and method extensions, the Aspectual Mixins and Aspectual Mixin Layers can solve all stated problems. However, the Aspectual Mixin approach yields some problems regarding the instantiation and life time. Moreover, it is currently not clear if the mixture of aspect and Mixin subsets leads to deeper problems. Currently, Aspectual Mixin Layers are the only implemented variant (see [2]).

A further highlight of all three AOP extensions is a specific bounding mechanism that supports a robust incremental design. Originally it was proposed by Lopez-Herrejon and Batory [21]. They argue that with regard to program family evolution features should only affect features of prior development stages. Current AOP languages, e.g. AspectJ and AspectC++, do not follow this principle. This decreases aspect reuse and complicates incremental design. Consequently, our three extensions follow this principle. To achieve this bounding mechanism, the user-declared join point specifications must be restructured: Type names in wildcards are translated to match only the types of the current and the parent layers. Each wildcard expression that contains a type name is translated into a set of new expressions that refer to all type names of the parent classes. Figure 11 shows a synchronization aspect that is part of an Aspectual Mixin Layer. It has two parent layers (*Base*, *Log*) and several child layers. FeatureC++ transforms the aspect and the pointcut as depicted in Figure 12. This transformation works similar for Aspectual Mixins. In case of Multi Mixins we have to add a mechanism for combining wildcard expression logically. Unfortunately, we have no formal evidence that this transformation does not capture inadvertently classes of later development stages. This is part of future investigations.

Finally, we want to emphasize that all three approaches are not specific to FeatureC++. All concepts can be applied to other FOP or AOP languages. We

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

Fig. 11. A simple pointcut expression.

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

Fig. 12. Transformed pointcut.

have implemented a first prototype of FEATUREC++ including the support for Aspectual Mixin Layers. The implementation is described in [2] and a prototype is available at our web site.

5 A Case Study

This section introduces a case study that gives an overview of the functionality of FeatureC++. We choose the stock information broker example, adopted from [24], in order to point to the benefits of Aspectual Mixin Layers compared to common FOP approaches. We show how FEATUREC++ overcomes the problems discussed in Section 4. The case study was implemented using our prototype.

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. 13). 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.

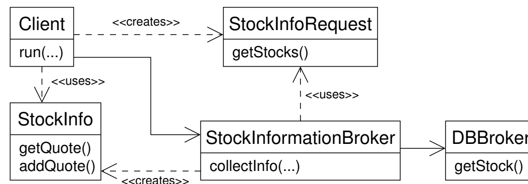

Fig. 13. Stock Information Broker.

All classes are encapsulated in a Mixin Layer. In other words, this Mixin Layer implements a basic stock information broker feature (*BasicSIB*). Figure 14 shows a relevant subset of this feature.

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 15 depicts this feature implemented using common FOP concepts. *Client* is refined by an account management (Lines 16-22), *SIR* is refined by a price calculation (Lines 2-5), and *SIB* charges the account when passing information to the client (Lines 10-12).

There are several problems to this approach: (1) The *Pricing* feature is expressed in terms of the structure of the *BasicSIB* feature. This problem is caused by the inability of FOP to express non-hierarchy-conform refinements. It would

```
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 };
```

Fig. 14. The basic stock information broker (BasicSIB).

be better to describe the *Pricing* feature using abstractions as product and customer. (2) The interface of *collectInfo* was extended. Therefore, the *Client* must extend 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 of clients cannot be dynamically altered, e.g. depending on the runtime control flow. Moreover, it is assigned to the *SIB* which is clearly not responsible for this function.

```
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 };
```

Fig. 15. The pricing feature using FOP (Pricing).

Pricing Feature as Aspectual Mixin Layer. Figure 16 depicts the *Pricing* feature implemented by an Aspectual Mixin Layer. The key difference is the *Charging* aspect. It intercepts calls to the method *collectInfo* (Lines 2-4) and charges the calling client depending on its request (Lines 5-6). 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 clients reference to the extended *collectInfo* method (not depicted). In both cases, the *Client* does not need to extend the *run* method.

A further advantage is that the charging of client's accounts can be made dependent to the control flow (using the *cflow* or *if* pointcut). This makes it possible to implement the charging function variable. Finally, our example shows that by using Aspectual Mixin Layers we have to refine only these classes that play the roles of product (*SIR*) and customer (*Client*).

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

Fig. 16. The pricing feature using Aspectual Mixin Layers (Pricing).

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 elegantly handle dynamic crosscutting, interface extensions, and non-hierarchy-conform refinements. Furthermore, Aspectual Mixin Layers can modularize homogeneous crosscuts and prevent excessive method extensions by using aspects (not shown). Due to the lack of space a description of a logging feature (homogeneous concern) that extends the broker application at multiple join points (preventing excessive method extensions) is omitted. The implementation is straightforward and was described many times. Table 1 summarizes the contribution of Aspectual Mixin Layers.

We readily admit that this simple case study cannot prove our ideas, and we do not intend to do so. This case study serves as proof of concept only and has the aim to ease the understanding of our ideas. Mature case studies are supposed to flesh out our theses in future work.

problem	solution	example
homogeneous crosscuts	pointcuts and advices	logging code is included in a set of methods
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 their runtime state
method extensions	wildcards in pointcut expressions	match all methods with price transfer

Table 1. Advantages of FeatureC++ Aspectual Mixin Layers.

6 Related Work

Work in several fields is related: programming support for incremental designs, AOP-related techniques, and the combination of AOP and FOP.

Programming support for incremental designs. One appropriate way to implement features of program families in a modular way are Mixin Layers [31]. Mixin Layers can be implemented using C++ templates [31], *P++* [29], *Jak* [5], *Java Layers* [8], *Jiazzi* [23], and *Delegation Layers* [25]. All these approaches leave aside the problem of lacking crosscutting modularity.

The constructor problem in incremental designs was introduced by Smaragdakis et al. [30]. Java Layers solve it by automatic constructor propagation from parent to child classes [8]. Eisenecker et al. utilize static C++ metaprogramming [13]. Several approaches solve the extensibility problem, introduced by Findler et al. [14]: Java Layers [8], Jak [5], Jiazzi [23]. Regarding the constructor problem and extensibility problem, FeatureC++ is inspired by these approaches.

Aspects and separation of concerns. [24, 19–21] discuss the drawbacks of current aspect-oriented languages, in particular no module boundaries, no feature cohesion, etc. FeatureC++ overcomes these problems by combining FOP and AOP concepts. This increases the crosscutting modularity and feature cohesion. Further, this preserves clear module boundaries and allows to scope aspect bindings.

Hyper/J supports multi-dimensional separation of concerns for Java [32]. This approach to software development is more general than that of FeatureC++ because it addresses the evolution of all software artifacts, e.g., documentation, makefiles, etc. However, Hyper/J has a lot of similarities to AHEAD [4]. Since FeatureC++ can be embedded into AHEAD it is an appropriate complement to Hyper/J.

The *Law of Demeter for Concerns (LoDC)* states that concerns should only know other concerns that contribute to its own functionality [18]. Following this principle (1) eases the incremental evolution of software by adding concern by

concern and (2) minimizes the number of feature interactions. FeatureC++ follows LoDC and enables a clear encapsulation of concerns. The supported bounding mechanism scopes aspects in order to reduce unpredictable feature interactions.

Classpects combine capabilities of aspects and classes to unify the design of layered module systems [28]. They are related to Aspectual Mixins, whereas classpects unify advices and method bodies (advices can be explicitly invoked), but do not support mixin-based refinements.

AspectJ-like languages can express Mixins too. Using static introductions, several classes (and methods) can be refined. In the face of heterogeneous crosscuts, for each target class a new aspect must be introduced. Otherwise, one aspect declares all introductions. The problem of the first approach is that it does not support feature cohesion. Moreover, the target classes are defined at development time. Therefore, an easy exchange of the target layers is not possible (because class names change which is not the case with Mixins). The second approach merges multiple refinement chains into one aspect. This may destroy the logical structure. Furthermore, our Multi Mixins can be seamlessly integrated into Mixin Layers and support the FOP paradigm. Moreover, they support incremental development by a novel bounding mechanism (see Sec. 4.1).

Aspects, Features, and Collaborations Mezini et al. show that using AOP as well as FOP standalone lacks crosscutting modularity [24]. 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. Lieberherr et al. [19] introduce *Aspectual Collaborations* that encapsulate aspects into modules with expected and provided interfaces. The main focus is similar to CaesarJ.

Kendall explores the connection between role modeling and AOP [15]. However, she does not consider the embedding of aspects into collaborations. Furthermore, her approach has several drawbacks regarding cohesive role refinements.

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 [10]. They distinguish between orthogonal and weak-orthogonal features/concerns.

Loughran et al. support the evolution of program families with *Framed Aspects* [22]. They combine the advantages of frames and AOP in order to serve unanticipated requirements. Frames are related to Aspectual Mixins and Aspectual Mixin Layers. Both allow to parameterize aspects at instantiation time.

7 Conclusion

This paper has presented FeatureC++, a novel FOP language extension to C++ that additionally adopts AOP concepts. Besides common FOP concepts it supports several useful C++-specific extensions, e.g. multiple inheritance, generic programming. After an introduction to FeatureC++, this paper contributes a summary of several weaknesses of FOP in modularizing crosscutting

concerns. We have explained how the weaknesses lead to problems in implementing incremental designs. Consequently, we propose three approaches to solve these problems, in particular, Multi Mixins, Aspectual Mixin Layers, and Aspectual Mixins. All these approaches adopt language concepts of AOP. A further highlight is a special bounding mechanism that supports a robust incremental development of program families. All three approaches are completely independent of FeatureC++ and can be applied to other FOP/AOP languages. Currently, we have implemented a prototype of FeatureC++ that supports most of the discussed language features, including Aspectual Mixin Layers. One can download a preliminary version of FeatureC++ at our web site[7]. Our case study has shown that FeatureC++ with its AOP extensions is able to elegantly express dynamic crosscutting, homogeneous crosscuts, non-hierarchy-conform refinements, and to cope with excessive method extensions and interface extensions.

In future work we want to investigate further in the relationship and symbiosis of FOP and AOP. In particular, we are interested in refining and evolving pointcuts and advices, as well as in different bounding mechanisms. Furthermore, we plan to implement and evaluate Multi Mixin and Aspectual Mixins. More complex case studies shall prove our results.

8 Acknowledgments

We would like to thank Don Batory and Erik Buchmann, as well as the program committee for comments on drafts of this paper.

References

1. S. Apel and K. Böhm. Towards the Development of Ubiquitous Middleware Product Lines. In *ASE'04 SEM Workshop*, volume 3437 of *LNCS*. Springer, 2005.
2. S. Apel et al. FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technical report, Department of Computer Science, Otto-von-Guericke University, Magdeburg, Germany, 2005.
3. D. Batory et al. Creating Reference Architectures: An Example from Avionics. In *Symposium on Software Reusability*, 1995.
4. D. Batory, J. Liu, and J.N. Sarvela. Refinements and Multi-Dimensional Separation of Concerns. *ACM SIGSOFT*, 2003.
5. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, 30(6), 2004.
6. D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems*, 9(2), 1997.
7. R. Cardone et al. Using Mixins to Build Flexible Widgets. In *AOSD*, 2002.
8. R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *ICSE*, 2001.
9. A. Colyer and A. Clement. Large-Scale AOSD for Middleware. In *AOSD*, 2004.

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

10. A. Colyer, A. Rashid, and G. Blair. On the Separation of Concerns in Program Families. Technical report, Computing Department, Lancaster University, 2004.
11. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.
12. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
13. U. W. Eisenecker, F. Blinn, and K. Czarnecki. A Solution to the Constructor-Problem of Mixin-Based Programming in C++. In *Workshop on C++ Template Programming*, 2000.
14. R. Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *ICFP*, 1998.
15. E. A. Kendall. Role Model Designs and Implementations with Aspect-Oriented Programming. In *OOPSLA*, 1999.
16. G. Kiczales et al. Aspect-Oriented Programming. In *ECOOP*, 1997.
17. R. Laddad. *AspectJ in Action – Practical Aspect-Oriented Programming*. Manning Publication Co., 2003.
18. K. Lieberherr. Controlling the Complexity of Software Designs. In *ICSE*, 2004.
19. K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal (Special issue on AOP)*, 46(5), 2003.
20. R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, 2005.
21. 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.
22. N. Loughran et al. Supporting Product Line Evolution with Framed Aspects. In *AOSD ACP4IS Workshop*, 2004.
23. S. McDirmid and W. Hsieh. Aspect-Oriented Programming in Jiazzi. In *AOSD*, 2003.
24. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT*, 2004.
25. K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *ECOOP*, 2002.
26. K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. In *ECOOP*, 2005.
27. D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE TSE*, SE-5(2), 1979.
28. H. Rajan and K. J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *ICSE*, 2005.
29. V. Singhal and D. Batory. P++: A Language for Large-Scale Reusable Software Components. In *Workshop on Software Reuse*, 1993.
30. Y. Smaragdakis and D. Batory. Mixin-Based Programming in C++. In *GCSE*, 2000.
31. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM TOSEM*, 11(2), 2002.
32. P. Tarr et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, 1999.