An Object-Oriented Approach to Feature Interaction

Christian Prehofer,
Institut für Informatik,
Technische Universität München,
80290 München, Germany,
http://www4.informatik.tu-muenchen.de/~prehofer

Abstract

In this paper, we address the feature interaction problem as a software composition problem. We show the applicability of a new programming style, feature-oriented programming (FOP), to typical feature interaction problems. FOP generalizes inheritance as in object-oriented programming and allows to compose objects with individual services from a set of features. In particular, we discuss exception handling within FOP and apply FOP to automata-based description techniques.

1 Introduction

In this paper, we address the feature interaction problem as a software composition problem. The recent interest in feature interactions, mostly stemming from multimedia applications [13, 2, 4], shows that there is a large demand for expressive composition concepts where objects with individual services can be created.

In our approach, objects with individual services can be created from a set of features. For this composition, we employ a feature composition architecture, which is a generalization of inheritance in object-oriented programming. Whereas the latter allows for incremental change, we compose features and their interactions in a similar, but more refined way.

With feature-oriented programming (FOP), we can solve two frequent feature interaction problems, which occur not only in telecommunication examples:

- A typical problem is that one feature has to be adapted in the presence of another one. The idea of the interaction handling is to lift one feature into the context of the other by adapting its services. This lifting into a context is the central idea of the composition architecture, which generalizes inheritance in object-oriented programming.

- Often, one has to give one feature precedence over the other, if their resources or impacts overlap. This is achieved by an ordering of features, given (implicitly) by the interaction handling.

In the following section, we present feature-oriented programming as an extension of the language Java [5]. Java is particularly suitable for our approach, but most other object-oriented languages would do as well. More details and definitions of FOP can be found in [8, 9].

After an introduction to the feature model, we show two applications, which both require some extensions and adoptions of this approach. The first example in Section 3 presents the vanilla example of feature interactions, call forwarding and call screening. For this, we develop a set of features which can be composed (almost) arbitrarily. We use exception handling as in Java for our examples and show how it integrates with our feature model.

The second example in Section 4 examines how a graphical description technique can be modeled via features. We consider simple hierarchical, finite-state automata, which often serve as the basis of the software design. Thus, we show composition techniques for automata and address interaction problems, which are difficult to model with simple automata.

2 Introduction to Feature Oriented Programming

In the following, we introduce FOP [8] by comparing it to object-oriented programming. A feature declares instance variables and defines methods, similar to abstract subclasses or mixins [1]. The main difference is that we separate the core functionality of a subclass from overwriting methods of the superclass into different entities.

We resolve feature interactions by lifting functions of one feature to the context of the other. Similar to inheritance, this is accomplished by adapting (overwriting) methods of the lifted class. Lifters depend on two features and are separate entities used for composition. In contrast, inheritance just overwrites methods of the superclass.

FOP allows to compose objects from individual features (or abstract subclasses) in a fully flexible and modular way. Its main advantage is that objects with individual services can be created just by selecting the desired features, unlike object-oriented programming. The main novelty of this approach is a modular architecture for composing features with the required interaction handling, yielding a full object. As we only compose objects, there is no real notion of a class, which is hence often confused with the (type of) objects.

Consider for instance an example modeling stacks with the following features:

Stack, providing push and pop operations on a stack.

Counter, which adds a local counter (used for the size of the stack).

Lock, adding a switch to allow/disallow modifications of an object (here used for the stack).

In an object-oriented language, one would extend a class of stacks by a counter and then extend this with lock. Usually, a concrete class is added onto another concrete class. We will generalize this to independent features which can be added to any object. For instance, we can run a counter object with or without lock. Furthermore, it is easy to imagine variations of the features, for instance different counters or a lock which not even permits read access. With FOP, we show that it is easy to provide such a set of features with interaction handling for simple reuse.

In general, a set of features replaces the rigid structure of conventional class hierarchies. The composition of features uses an architecture for adding interaction resolution code (via overwriting) which is similar to constructing a concrete class hierarchy. To construct an object (or class type), features are added one after another in a particular order. In addition, if a feature is added to a combination of n features, we have to apply n lifters in order to adapt the inner n features.

Feature oriented programming is advantageous for the following reasons:

- It yields more flexibility and clarifies dependencies between features (or classes), as objects with individual services can be composed from a set of features. This is desirable, if there are many alternative features/classes in a framework or if new functionality has to be incorporated.

- As the core functionality is separated from interaction handling, it provides more structure, clearer interfaces and encourages to write independent, reusable code. In other words, many subclasses should be an independent entity, and not a subclass.

Let us continue with the example modeling variations of stacks. We first define Java interfaces for features. Although this is not strictly needed for our ideas they are useful if there are several implementations for one interface.

```
interface Stack {
  void empty();
  void push( char a);
  void push2( char a);
  void pop();
  char top();
}

interface Counter {
  void reset();
  void inc();
  void dec();
  int size();
}

interface Lock {
  void lock();
  void unlock();
  boolean is_unlocked();
}
```

The code below provides base implementations of the individual features. It is presented as a simple extension of Java. The notation `feature SF` defines a new feature named `SF`, which implements stacks. Similar to class names in Java, `SF` is used as a new constructor. Using the other two feature implementations, `CF` and `LF`,

```
new LF (CF (SF))
```

creates an object with all three features. For interaction handling, it is important that features are composed in a particular order, e.g. the above first adds `CF` to `SF` and then adds `LF`.

```
feature SF implements Stack {
   String s = new String();
   void empty() { s = "";}  // Use Java Strings ...
   void push( char a) {s = String.valueOf(a).concat(s); };
   void pop() {s = s.substring(1); } ;
```

```
   char top() { return (s.charAt(0) ); } ;
   void push2( char a) {this.push(a) ; this.push(a); };
}

feature CF implements Counter {
   int i = 0;
   void reset() {i = 0; };
   void inc() {i = i+1; };
   void dec() {i = i-1; };
   int size() {return i; };
}

feature LF implements Lock {
   boolean l = true;
   void lock() {l = false;};
   void unlock() {l = true;};
   boolean is_unlocked() {return l;};
}
```

In addition to the base implementations, we need to provide lifters, which replace method overwriting in (abstract) subclasses. Such lifters are separate entities and always handle two features at a time. The following code lifts feature interfaces over concrete feature implementations. For instance, `feature CF lifts Stack` adapts the functions of `Stack` to the context of `CF`, i.e. the counter has to be updated accordingly. When composing features, this lifter is used if `CF` is added to an object (type) with a feature with interface `Stack`, and not just directly to a stack implementation. This is important for flexible composition, as shown below.

```
feature CF lifts Stack {
   void empty() {this.reset(); super.empty();};
   void push( char a) {this.inc(); super.push(a);};
   void pop() { this.dec(); super.pop();};
}

feature LF lifts Stack {
   void empty() {if (this.is_unlocked() ) {super.empty();}};
   void push( char a)
        {if (this.is_unlocked() ) { super.push(a);}};
   void pop() { if (this.is_unlocked() ) { super.pop();}};
}

feature LF lifts Counter {
   void reset() {if (this.is_unlocked()) {super.reset();}};
   void inc() {if (this.is_unlocked()) {super.inc();}};
   void dec() {if (this.is_unlocked()) {super.dec();}};
}
```

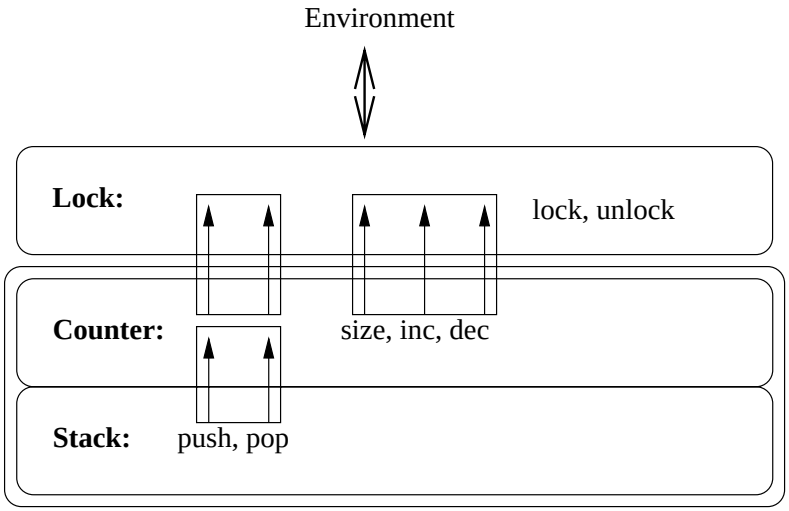

Figure 1: Composing features (rounded boxes) by lifters (boxes with arrows)

Note that methods which are unaffected by interactions are not explicitly lifted, here `top` and `size`.

This modular specification of features, separated from their interactions, allows the following object compositions:

- Stack with counter
- Stack with lock
- Stack with counter and lock
- Counter with lock

For all these combinations, the three lifters shown above adapt the features to the corresponding combination. The resulting objects of this type behave as desired. In addition, we can of course use each feature individually (even the lock). Furthermore, it is easy to imagine different variations of these features as well as additional ones.

The composition of lifters and features is shown in Figure 1 for an example with three features. To compose stack, counter, and lock, we first add the counter to the stack and lift the stack to the counter. Then the lock feature is added and the inner two are lifted to lock. Hence the methods of the stack are adapted again, using the lifter from stack to lock. In general, when adding a feature b to an object having the set of features A, the methods of each feature in A are lifted individually to the new context. Note that the composed object provides for the functionality of all selected features, but for composition with lifters we need an ordering. For instance, the outermost feature is not lifted, similar to the lowest class in a class hierarchy, whose functions are not overwritten (see also [8]).

Although inheritance can be used for such feature combinations, all needed combinations, including feature interactions, have to be programmed explicitly. In contrast, we can (re)use features by simply selecting the desired ones when creating an object. Similarly, if abstract classes are used, the interactions have to be programmed by hand.

For a more detailed definition of features, we refer to [8]. This paper shows two simple translations of our feature-based language extension of Java into Java, one via inheritance and one via aggregation and delegation.

3 Feature Interaction in Telecommunications

A well known problem in feature interaction stems from the abundance of features telephones (will) have. For instance, consider the following conflict occurring in telephone connections: B forwards calls to his phone to C. C screens calls from A (ICS, incoming call screening). Should a call from A to B be connected to C? In this example, there is a clear interaction between forwarding (FD) and ICS, which can be resolved in several ways. For many other examples we refer to [3].

We demonstrate our techniques with the following set of features for this domain of connecting calls:

- Forwarding of calls
- ICS (incoming call screening)
- OCS (outgoing call screening)

Although ICS and OCS look very similar, there are small differences. For instance, they may interact differently with other features, as shown below.

3.1 The Basic Phone

The basic building block is a feature `Phone`, which provides a function `connect` for computing the phone reached by some dialed number. In addition, we use a simple technique for adding the actual services to the full object. Each feature adds its functionality by extending a function `dispatch` (for feature dispatch), which is used by `connect`. As we use exceptions for modeling a busy signal, the function `dispatch` may throw an exception, as shown in the following Java interface description:

```
interface Phone {
  int connect(int dest) ;
  int dispatch(int dest) throws Busy ; }
```

The implementation provides for a trivial connect functionality, just in order to set the stage for other features. Note that we need an explicit constructor `Ph` here, which initializes the instance variable used for the origin of a call. (In Java, class names are used as self-definable constructor functions which may also have arguments.)

```
class Ph implements Phone {
        // origin
  int o;
        // function for creating and initializing objects
  Ph(int orig) { o = orig;} ;
        // trivial dispatch
  int dispatch(int dest) throws Busy
     {return dest;};
```

```
  int connect(int dest) {
    try{        return dispatch(dest) ;    }
    catch(Busy B) {  println("Busy " ); return 0; } }
}
```

3.2 Forwarding

Next we add a feature for forwarding with the following interface:

```
interface Forward {
  boolean fd_check(int i);
  int forward(int dest);  }
```

The code is again as simple as possible, just forwarding selected numbers to the next number via an auxiliary function `fd_check`.

```
feature Fd implements Forward  {
     // aux. function
  boolean fd_check(int i)
         // naive check if forwarding is desired
     { return (i == 5 || i == 7 || i == 10 ||
               i == 11 || i == 12); };
  int forward(int i) { return i+1; };
}
```

To integrate the service, we lift the `dispatch` function in the following lifter:

```
feature Fd lifts Phone {
  int dispatch(int dest) throws Busy {
     if (fd_check(dest))
               //  recursive forwarding !
          return connect(forward(dest));
     else return super.dispatch( dest);
   }
}
```

Note that the above either calls `super.dispatch` in order to invoke (possibly) other features, or calls `connect` to attempt a recursive connect attempt. (This recursive forwarding is not limited, for simplicity.)

3.3 Incoming Call Screening

For ICS we use a simple check for each call and raise the exception busy if ICS disallows a call. The structure of the following code is as above.

```
interface IcsI {
   int ics(int dest) throws Busy ;
}

feature Ics implements IcsI {
  Ics(int orig) {super(orig); };
```

```
    // aux. functions
  boolean ics_check(int i) {
            // no calls from 5 to 8
    return (o == 5 & i == 8);  };

  int ics(int dest) throws Busy {
    if (ics_check(dest)) throw new Busy();
    else return dest;};
}

feature Ics lifts Phone {
  // lift Phone
  int dispatch( int dest) throws Busy {
            // add ICS service
       return super.dispatch( ics(dest) ) ;};
}
```

3.4 Resolving the ICS/Forward-Interaction

To resolve the interaction between forwarding and ICS, we lift the forward function to ICS. We chose the lifting for forward as follows:

```
feature Ics lifts Forward {
    // lift forward
  int forward(int dest) {
            // update origin (also ok if not forwarded!)
       o = dest;
       return super.forward(dest); };
}
```

In case of forwarding over several hops, ICS is checked for each (intermediate) hop wrt the next hop. If only a check wrt the origin of the call is desired, one just has to adapt the lifter (and not update the origin of the call). Thus, lifting allows for a modular resolution the interaction between two features.

3.5 Outgoing Call Screening

OCS is quite similar to ICS, but we model interaction with forwarding differently: OCS should always be checked from the initial phone to the final destination. With this choice OCS can be modeled similar to ICS, but with a different interaction code. The code is not shown here for brevity.

An even stronger condition would be to check OCS for each intermediate hop. For this, one needs an extra data structure for storing these.

3.6 Examples

It is now possible to create objects with any subset of the three features ICS, OCS and FD. For instance, with the object `con` created by

```
con = new Ics (FD (Ph 5)))
```

Fd and ICS are selected and the originating phone is set to 5.

With the above code and settings, we obtain the following examples for a connect call from phone 5 with all features:

```
   // just gives 6
 println( con.connect(5));
   // just gives 6
 println( con.connect(6));
   // gives 8, as forwarded to 8, which is allowed by ICS
 println( con.connect(7));
   // gives Busy, as forwarded to 8,
      which is not allowed by ICS
 println( con.connect(8));
   // gives 13, as forwarded twice
 println( con.connect(11));
```

It is easy to imagine other features and also variations of the above features. Our approach allows to compose such features in a flexible way. This provides for a clear structure of their dependencies, which is needed if the number of complementary or alternative features grows.

3.7 Features and Exceptions

This example shows that FOP easily accommodates exceptions as in Java. Java requires to declare an exception for any method whose body may raise one, and also in interfaces for which one implementation raises an exception. This rigid and fully explicit declaration of exceptions in Java can be used for feature combination in FOP, as all affected methods are marked.

On the other hand, this restrictive scheme may be a bit too inflexible. For instance, in the above example, we have to declare an exception for the function `dispatch`, although a concrete feature composition may not use exceptions. In such a case, it might be desirable to handle exceptions in liftings, such that a feature can be adapted to a context where an exception is not declared (and handled). It should also be remarked that adding and removing exceptions in Java and hence in this setting can be quite tedious.

4 Features and Automata

We study in the following an example with was studied by Teege in [11, 12] as a subset of a group-ware application. In this example, hierarchical automata are used to describe the status of a document. As discussed in [11], there arise feature interactions, which are not easily modeled by simple automata concepts. We show here how to model automata by feature-oriented programming. The main benefit we gain is composition of automata, where feature interactions are handled as in FOP. We show that the interactions in this application domain can be modeled appropriately with FOP. Hence FOP provides the basis for developing more advanced automata composition concepts, which possibly permit graphic notation. This is discussed below.

The idea of this example is to specify smaller automata as features and to compose them with the techniques of FOP. This flexible composition is the main difference to the many other techniques for graphic description of software components.

We model the following features as automata:

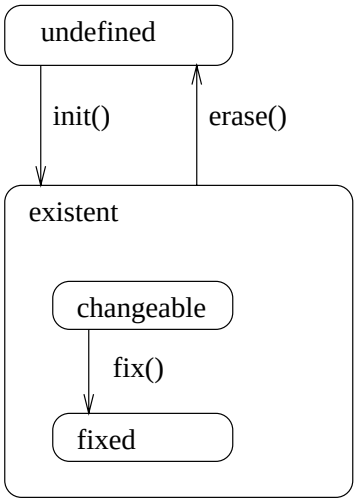

Figure 2: Hierarchical Automaton for the Status of a Document

- DocAut(omaton) with states `existent` and `undef`, and a transition (or function) `init`.
- ChangeAut with states `changeable` and `fixed` and a transition `fix`. ChangeAut assumes DocAut and defines a sub-automaton of state `existent`. This can be viewed as a refinement of a state.
- ErasableDoc assumes also DocAut and defines a function `erase`.

The combination of these three features is shown in Figure 2. The full application contains a larger set of features. The motivation for this example is to select the desired features individually for each document.

The interaction problem of this example is clearly between ErasableDoc and ChangeAut. If ChangeAut is in state `fixed`, then erase should not be possible. This is achieved here by lifting ErasableDoc to ChangeAut. As shown below, we disallow erase if the local state is `fixed`.

Note also that hierarchical automata pose another problem which can be viewed as an interaction. The problem arises, since a sub-automaton is only active, if the global automaton is in the corresponding, refined state. In case of a global transition to a new state, which has been refined to a sub-automaton by some other feature, it is unclear if the sub-automaton should be (re-)initialized or if its old state be preserved.

The following code defines the base document with two global states. The feature implementation requires another feature BA, as indicated with the requires clause. Feature BA allows DA, the implementation of DocAut, to use some basic functionality of BA to construct automata. (This allows to use the method `newstate` in order to define a new state. Its details shall not be discussed here.) In general, the base functionality of a new feature can rely on the functionality of the required ones.

```
interface DocAut {
  void init();
```

```
  final int undef;
  final int existent;
}

feature DA implements DocAut requires BA {
  final int undef = newstate();
  final int existent = newstate();
        // default state
  DA() {state = undef;};

  void init() { if (state==undef) state = existent; };
}
```

The second component, ErasableDoc, just adds one transition. As it adds no states, its implementation requires DA in order to implement the method. The lifting of DocAut is trivial (and could be omitted).

```
interface ErasableDoc   {
   void erase();
}

feature EA implements ErasableDoc requires DA {
   void erase() { if (state==existent) state = undef; };
}

feature EA lifts DocAut {
   void init() {super.init(); };
}
```

The definition of ChangeAut is more involved, since it defines a local automaton, named `local_aut`, with two states. Its constructor CA initializes its local state.

```
interface ChangeAut {
  void fix();
}

feature CA implements ChangeAut requires BA {
     // CA refines a state, hence uses a local automaton
   BA local_aut = new BA();
   final int changeable = local_aut.newstate();
   final int fixed = local_aut.newstate();
        // initialize with default state
   CA() {super(); local_aut.state = changeable;};

   void fix() { local_aut.state = fixed;};
}
```

The interesting aspects of ChangeAut are defined in its lifters. To lift DocAut, we turn the local automaton into an initial state, if `init` is invoked. Furthermore, to solve the interaction between with ErasableDoc, we redefine `erase`.

```
feature CA lifts DocAut {
   // lift DA: initialize
   void init() {
                // changeable is default state
        local_aut.state = changeable;
        super.init(); };
}

feature CA lifts ErasableDoc {
   // lift EA:
   void erase() {
                // disable erase if fixed
        if (local_aut.state != fixed) super.erase(); };
}
```

For the full example with more, but similar interactions, we again refer to [12]. Note that our simple implementation of automata is sufficient for our purpose. For pure automata, there clearly exist more efficient implementations. However, for many software applications, automata just pose as a skeleton of the actual program, which is completed as a further step. For this purpose, our implementation is more appropriate, as it is extensible.

4.1 Discussion

We have shown that typical composition problems of automata descriptions can be modeled by FOP. Clearly, when working with automata, a graphical notation for interaction resolution is desirable. For some of the typical interaction cases it is possible to define graphical equivalents. For such advanced specification concepts with automata, we refer to [6, 11].

For instance, the two interactions above are supported by some special purpose languages with graphical notation. The first problem of disabling a transition is possible in an object-oriented variant of Statemate [6] and in Hierastates [12]. In both languages, the local transition has precedence over the global.

The other problem regarding the life-time of local states is resolved by particular annotations in Statemate [6]. As a typical problem of such simple notations is that certain aspects cannot be modeled, e.g. that only particular global transitions reset the local state.

Our main point here is that common automata concepts allow for typical composition operations on automata, but do not consider interactions between components explicitly, as done here. With the concepts developed here, it is possible to compose just the wanted features/automata, where interactions are resolved as specified. It remains for future work, to extract a broad set of typical interactions and to devise graphical notation for them.

5 Conclusions

We have shown that feature-oriented programming allows a fully modular and compositional modeling of typical feature interaction problems. This includes common extensions and tools, such as exception handling and graphic descriptions with automata.

While much research has been devoted to feature interaction detection and specification, we believe that flexible composition techniques can also support the development and evolution of large and complicated software. As feature-oriented programming is an extension of the object-oriented programming paradigm, we argue that object-oriented programming can benefit from the ideas developed for feature interactions. Further work should also focus on more elaborated composition techniques which allow to consider interaction between between several agents or objects. For other extensions such as parameterized features and a detailed comparison to object-oriented techniques we again refer to [8].

Among related work on software composition is adaptive programming [7], which aims at adaptable software design. Some versions of adaptive programming [10] apply similar techniques as we do here, but do not consider the idea of feature interaction and composition.

References

[1] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, San Francisco, April 1992. IEEE Computer Society.

[2] E.J. Cameron, N. Griffeth, Y.-J. Linand M.E. Nilson, W.K. Schnure, and H. Velthuijsen. A feature-interaction benchmark for IN and beyond. *IEEE Communications Magazine*, 31(3):64–69, March 1993.

[3] E.J. Cameron, N. Griffeth, Y.-J. Linand M.E. Nilson, W.K. Schnure, and H. Velthuijsen. A feature interaction benchmark for in and beyond. In L. G. Bouma and Hugo Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 1–23, Amsterdam, 1994. IOS Press.

[4] K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications III*. IOS Press, Tokyo, Japan, Oct 1995.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, September 1996.

[6] D. Harel and A. Naamad. The Statemate Semantics of Statecharts. *IEEE Transactions on Software Engineering Method*, 1996.

[7] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[8] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP '97*, 1997. To appear in Springer-LNCS.

[9] Christian Prehofer. From inheritance to feature interaction. In Max Mühlhäuser et al., editor, *Special Issues in Object-Oriented Programming. ECOOP 1996 Workshop on Composability Issues in Object-Orientation*, Heidelberg, 1997. dpunkt-Verlag.

[10] Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of object behavior using context relations. In David Garlan, editor, *Symposium on Foundations of Software Engineering*, San Francisco, 1996. ACM Press.

[11] Gunnar Teege. Hierastates: Flexible interaction with objects. Technical report, TU München, TUM-I9441, 1994.

[12] Gunnar Teege. Hierastates: Supporting workflows which include schematic and ad-hoc aspects. In Michael Wolf and Ulrich Reimer, editors, *Proc. of 1st Int. Conf. on Practical Aspects of Knowledge Management PAKM'96*, 1996.

[13] P. Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, XXVI(8), August 1993.