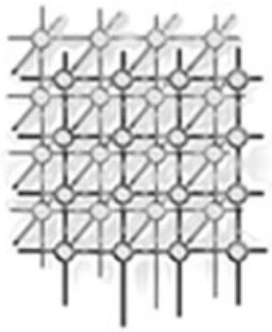
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2001; **13**:465–501 (DOI: 10.1002/cpe.583)

Object Systems



Feature-oriented programming: A new way of object composition‡

Christian Prehofer*,†

TU München, 80290 Munich, Germany

SUMMARY

We propose a new model for flexible composition of objects from a set of features. Features are services of an object and are similar to classes in object-oriented languages. In many cases, features have to be adapted in the presence of other features, which is also called the feature interaction problem. We introduce explicit interaction handlers which can adapt features to other features by overriding methods. When features are composed, the appropriate interaction handling is added in a way which generalizes inheritance and aggregation. For a set of features, an exponential number of different feature combinations is possible, based on a quadratic number of interaction resolutions. We present the feature model as an extension of Java and give two translations to Java, one via inheritance and the other via aggregation. We show that the feature model interacts nicely with several common language extensions such as type parameters, exceptions, and higher-order functions. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: feature model; Java; object composition

1. INTRODUCTION

A major contribution of object-oriented programming is reuse by inheritance or subclassing. Its success and its extensive use have led to several techniques which increase flexibility (mix-ins [1,2], around-messages in Lisp [3], class refactoring methods [4]). Other approaches use different composition techniques, such as aggregation and (abstract) subclasses.

In this paper we propose a new model for object-oriented programming which generalizes inheritance and includes most of the above concepts. Instead of a rigid class structure, we propose writing features which are composed appropriately when creating objects. Features are similar to abstract subclasses or mixins [2]. Viewed from the outside, a feature is a service offered by an object,

*Correspondence to: C. Prehofer, Siemens AG, 81359 Munich, Germany.
†E-mail: Christian.Prehofer@mch.siemens.de
‡A preliminary version of this article appeared under the title 'Feature-Oriented Programming: A Fresh Look at Objects' at ECOOP 1997.

Received 22 January 2001

similar to the notion of an interface in Java [5], where a class can implement several interfaces[§]. The main difference to class hierarchies is that we separate the core functionality of a subclass from overriding methods of the superclass. We view overriding more generally as a mechanism to resolve dependencies or interactions between features, i.e. some feature must behave differently in the presence of another one.

We resolve feature interactions by lifting functions of one feature to the context of the other. A lifter, also called adaptor or modifier, adapts the functions of one feature in the presence of another one. Similar to inheritance, this is accomplished by method overriding, but lifters depend on two features and are separate entities used for composition. A lifter is a relation between two features. With conventional inheritance, one can just override methods of the superclasses.

Due to separation of concerns, our new model allows one 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; in object-oriented programming, only instances of existing classes can be created. Hence feature-oriented programming is particularly useful in applications where a large variety of similar objects is needed. Examples are generic frameworks or libraries which can be used by different applications in various ways or in application domains which by nature require many variations. For the former, we present a small example of a generic data structure; for a larger example we refer to [6]. The area of telecommunication software sparked the research on feature interactions; an example from this area is shown in Section 7.3. The main novelty of this approach is a modular architecture for composing features with the required interaction handling, yielding a full object.

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).

Bound, which implements a range check, used for the stack elements.

Undo, adding an undo function which restores the state as it was before the last access to the object.

In an object-oriented language, one would extend a class of stacks by a counter and proceed similarly with the other features. Usually, a concrete class is added onto another concrete class. We 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 our approach, we show that it is easy to provide such a set of features with interaction handling for simple reuse.

With feature-oriented programming, a feature repository replaces the rigid structure of conventional class hierarchies. Both are illustrated in Figures 1 and 2. The composition of features in Figure 2 uses an architecture for adding interaction resolution code (via overriding) which is similar to constructing a concrete class hierarchy. To construct an object, features are added one after another following a

[§]Note that we do not have a notion of classes here, only objects which implement features.

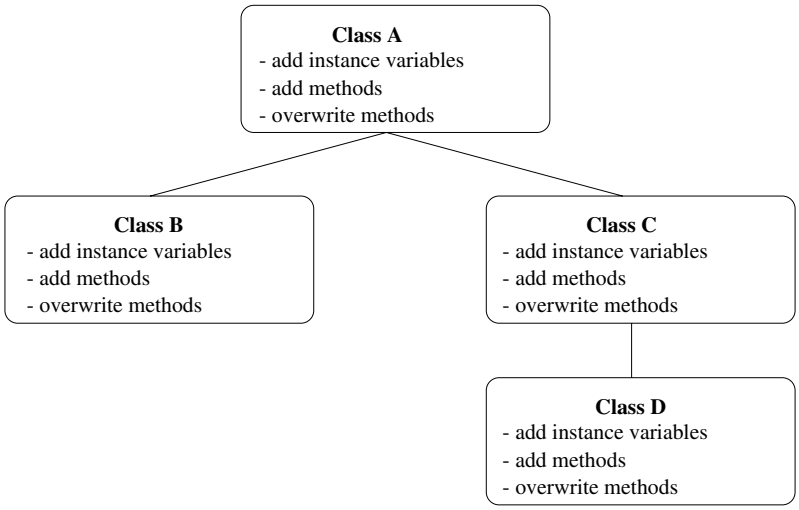


Figure 1. Typical class hierarchies.

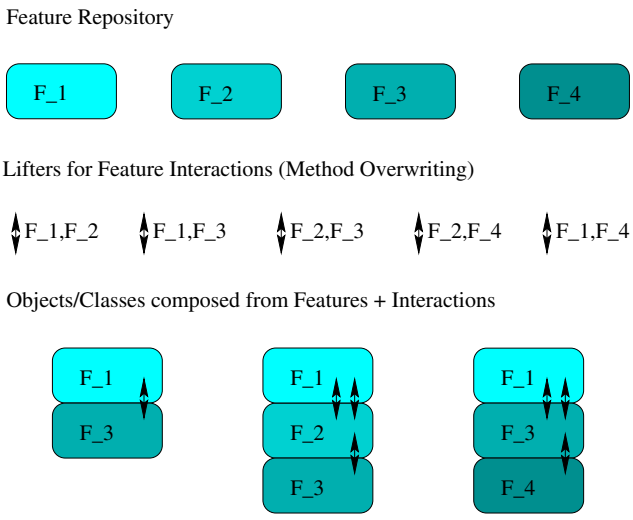


Figure 2. Composing objects in the feature model.

particular order, given by lifters[¶]. This follows the partial order on features, which is induced by the lifters since we disallow cycles. If a feature is added to a combination of n features, we have to apply n lifters in order to adapt the inner features.

As we consider interactions for two features at a time, there is only a quadratic number $\binom{n}{2} = (n^2 - n)/2$ of lifters, but an exponential number $\binom{n}{k}$, $k = 1, \ldots, n$, of different feature combinations can be created. For instance, in the above example, we have five features with 10 interactions and about 30 sensible feature combinations. This number grows if different implementations or variations of features are considered (e.g. single- or multi-step undo).

We show that feature-oriented programming generalizes object-oriented techniques and gives a new conceptual model of objects and object composition. A simple notion of features or services is used in Java in the form of interfaces. We use these to define the syntactic interface of features. For feature implementation, we add the idea of modular interaction handling and composition concepts. To show the relation to Java, we translate to Java [5] code for concrete feature selections, first using inheritance and then using aggregation and delegation. This explains the relations with known techniques and also compares both techniques. In fact, we will show two cases where aggregation is more expressive than inheritance, refining earlier results [7].

To summarize, feature-oriented programming is advantageous for the following reasons.

- It yields more flexibility, as objects with individual services can be composed from a set of features. This is clearly desirable if many different variations of one software component are needed or if new functionality has to be incorporated frequently.
- As the core functionality is separated from interaction handling, it provides more structure and clarifies dependencies between features. Hence it encourages to write independent, reusable code. In many cases subclasses should be independent entities, which can be reused in different places. This also makes class refactoring [4] much easier and sometimes unnecessary. The idea is similar to abstract classes, but we also cover dependencies between features.
- As we consider only liftings or interactions between two features at a time, the model is as simple as possible. Furthermore, a set of features can only be assembled in a particular order. We show that this composition scheme is sufficient in practice. Only in a few cases auxiliary features are needed.

Starting with this basic idea of features, there are several useful extensions. First, we show that parameterized features (similar to templates) work nicely with interactions and liftings (which replace inheritance). As we will see, there can also occur type dependencies between two features, which can be clearly specified in our setting. Another extension permits features to add exceptions which can be raised in lifters for a feature. Hence checking exceptional cases can be modeled as resolving an interaction. As liftings for some features are generic, we model this case via higher-order functions.

The main technical contributions and results in this paper are as follows.

[¶]As we only compose objects, there is no explicit class they belong to; instead, we often use the type of objects in a similar fashion.

- A language extension of Java for features and their interactions. For concrete feature compositions, we give translations into pure Java, one via inheritance and one via aggregation and delegation.
- An analysis of parameterized features and type dependencies between features, followed by a translation into Pizza [8], an extension of Java.
- The translations lead to a detailed comparison of aggregation and inheritance. This unveils two cases where aggregation is more powerful than inheritance due to typing problems.

A major premise of our approach is that interactions can be handled for two features at a time. This means that interactions between several features must be resolved by liftings between two features. We show that this is suitable for structuring software and is not a limitation in practice. Note that this is different from cases where some features are simply incompatible or contradictory. Another basic assumption is that features are composed in a linear order, although the interface appears to be a set of features. For instance, in some cases features have to be split into two separate features for this reason.

The origin of this idea of features goes back to applications of monad theory in functional programming, as discussed in [9–11]. In [10], composition of state monads is compared to inheritance and extended to other monads in functional programming. This did provide an experimental prototype for the language features presented here. On the application side, this work was motivated by the development in telecommunication and multimedia software, where feature interactions have recently attracted great attention [12,13].

In the following section, we discuss the first three features of the stack example. We define the feature-oriented extension of Java via translations in Section 3, followed by an extension to parameterized features in Section 4. This section also discusses the remaining two features, undo and bound. An extension to generic liftings via higher-order functions is shown in Section 5. Section 6 discusses features which introduce exceptions and raise exception in lifters.

Many examples from several areas are shown in Section 7, starting with variations of design patterns in Section 7.1. Section 7.2 models common software description techniques using automata by features and analyzes interactions. An interesting application area where feature interactions have been extensively examined are multimedia and telecommunication systems, as discussed in Section 7.3. This application area actually triggered this research. We conclude with discussions of the approach in Section 8.

2. FEATURE-ORIENTED PROGRAMMING BY EXAMPLE

In this section, we introduce feature-oriented programming using the above example modeling variations of stacks. (The undo and bound features are shown later in Section 4.) For this purpose, we present an extension of Java in the following.

We first define interfaces for features. Although not strictly needed for our ideas, they are useful if there are several implementations for one interface[‖].

```
interface Stack {
  void empty();
```

[‖] Furthermore, feature interfaces ease translation into Java, as a class can implement several interfaces in Java.

```
  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();
}
```

The code below provides base implementations of the individual features. Note that we only treat stacks over characters; parametric stacks will be considered later. 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 for a feature.

```
feature SF implements Stack {
   String s = new String();
                        // Use Java Strings to implement a Stack
   void empty() {s = ""; }
   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;};
}
```


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.

In addition to the base implementations, we need to provide lifters, which replace method overriding in subclasses. Lifters are separate entities, handling two features at a time. In the following code, features (denoted by their interfaces) are lifted over concrete feature implementations. For instance, the code below `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 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();}};
   int  size() { return super.size(); };
   char top() { return super.top(); } ;
}
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();}};
}
```

Methods which are unaffected by interactions are not explicitly lifted, e.g. top and size. Note that the lifting to the lock feature is schematic. Hence it is tempting to allow default lifters, as discussed in Section 5.

The modular specification of the three features, separated from their interactions, allows for the following object compositions:

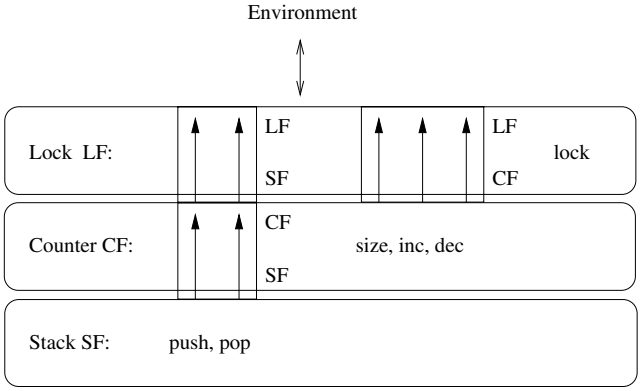


Figure 3. Composing features (rounded boxes) by lifters (boxes with arrows).

- stack with counter;
- stack with lock;
- stack with counter and lock; and
- counter with lock.

For all these combinations, the three lifters shown above adapt the features appropriately to the chosen combination. The resulting objects behave as desired. In addition, we can use each feature individually (even lock). With the remaining two features, bound and undo (shown later), many more combinations are possible in the same way.

The composition of lifters and features is shown in Figure 3 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 a typical class hierarchy which corresponds to the class of this object is usually drawn upside down compared to Figure 3.

The composed object simply provides the functionality of all selected features to the outside, but for composition we need an additional ordering. In particular, the outermost feature is not lifted, similar to the lowest class in a class hierarchy, whose functions are not overwritten.

Although inheritance can be used for such feature combinations, all needed combinations, including feature interactions, have to be assembled manually. In contrast, we can (re)use features by simply selecting the desired ones when creating an object.

In the above example, each feature can be used independently. In other examples it is often needed to write a feature assuming that some other feature is available. For this, a feature declaration may require other features, e.g. as in


```
feature DisplayAdapter assumes AsciiAdapter {
  void show_window(...) { ... }
  ...  }
```

Consequently, an implementation may use the operations provided by the feature `AsciiPrintable` in order to produce output on a window system.

In general, the base functionality of a new feature can rely on the functionality of the required ones. This idea of assuming other features is a further difference to conventional abstract subclass concepts. (Note that the extended object can obviously have more than just the required features.)

3. TRANSLATION TO JAVA

To provide a precise definition of our Java extension, we show two translations into Java. The first translation uses inheritance, while the second uses aggregation with delegation. Hence this also serves to compare the feature model with both of these approaches and will highlight two cases where both differ. The equivalence of the two translations is easy to see and similar to the formal comparison of delegation (or aggregation) and inheritance, presented in [7].

We assume the following abstract program with

- I_i feature interfaces,
- $I_i.t_{k_i}$ methods declared for interface I_i,
- F_i corresponding features,
- $F_i.vardecls$ declaration of instance variables,
- $F_i.f_{k_i}$ code for methods $I_i.t_{k_i}$,
- $F_{i,j}$ lifter for F_j to I_i, and
- $F_{i,j.f_{k_j}}$ code for lifting $I_j.t_{k_j}$ to I_i.

```
interface I1 {
                    // method declarations
    I1.t1;
    ⋮
    I1.tk1;
}
    ⋮
interface Im {
    Im.t1;
    ⋮
    Im.tkm;
}
feature F1 implements I1 assumes I1^l1, ..., I1^ln {
    F1.vardecls     // variable declarations
    I1.t1 F1.f1;    // method implementations
```

```
    ⋮
    I_1.t_{k_1} F_1.f_{k_1};
}
    ⋮
     // lifters
feature F_i lifts I_j {
    I_j.t_1 F_{i,j}.f_1;     // function redefinitions
    ⋮
    I_j.t_{k_j} F_{i,j}.f_{k_j};
}
```

We use this schematic program to translate concrete object creations into Java in two ways, inheritance and aggregation. For the translations, the feature interfaces are preserved, while the feature code is merged into concrete classes, as shown below.

For sake of presentation, the translation is simplified in order to make the obtained code as explicit as possible. Therefore, we assume that

1. the names of (instance) variables as well as method names are distinct for all features,
2. assume that method calls to *this* are explicit, i.e. always *this.fct*(...) instead of *fct*(...), and
3. variable declarations have no initializations.

3.1. Translation via inheritance

In this translation, we create a set of concrete classes, one extending the other, for each used feature combination $F_1(F_2(F_3(\ldots(F_n)\ldots)))$. For such a combination, we create n classes named $F_i\_F_{i+1}\_F_{i+2}\_\ldots\_F_n$ for $i = 1, \ldots, n$. These extend each other and add one feature after another. For instance, $F_1\_F_2\_F_3\_\ldots$ extends $F_2\_F_3\_\ldots$. The class $F_1\_F_2\_F_3\_\ldots$ adds the functionality for interface I_1 and lifters to F_1 for all others.

Formally, an object creation

`new` $F_1(F_2\ldots(F_n)\ldots)$

translates to

`new` $F_1\_F_2\_\ldots\_F_n$

This class is defined via the following Java classes for $i = 1, \ldots, n$:

```
class F_i_F_{i+1}_..._F_n extends F_{i+1}_..._F_n
    implements I_i, ..., I_n {
                  // Feature i implementation
    F_i.vardecls  // variable declarations
    I_i.t_1 F_i.f_1;  // function implementations
    ⋮
```


$I_i.t_{k_i}\ F_i.f_{k_i}$;
// Lift Feature i+1 to i
$I_{i+1}.t_1\ F_{i,i+1}.f_1$; // function redefinitions
⋮
$I_{i+1}.t_{k_{i+1}}\ F_{i,i+1}.f_{k_{i+1}}$;
⋮ // Lift Feature n to i
$I_n.t_1\ F_{i,n}.f_1$; // function redefinitions
⋮
$I_n.t_{k_n}\ F_{i,n}.f_{k_n}$;
}

The above schematic code implements feature i and, using lifters, possibly overrides the features in the extended class. A concrete example for the way lifters are composed is presented below. Recall also Figure 3 for this example. Observe that $n - i$ lifters are needed, which may call methods of the super class.

We have so far assumed that the features required for F_i via `assumes` are present in $F_{i+1}, \ldots, F_n$ for a feature combination $F_1(F_2 \ldots (F_i \ldots (F_n) \ldots)$. This restriction can be relaxed if we assume that the required ones are present in $F_1, \ldots, F_{i-1}$, similar to virtual functions which are only defined in a subclass. However, this does not work with this translation. The translation assumes that the features required for F_i via `assumes` are present in the extended class. In the class created for the combination $F_1, \ldots, F_i$, this however entails that undeclared identifiers occur in the translated code. This may only be allowed in a dynamically typed language with dynamic method lookup. Interestingly, this assumption is not needed for aggregation, which accounts for a small difference between the two translations. Another difference will be examined in the following section on parameterized features.

For instance, the three features from the introduction translate into the following class hierarchy, if an object of type `LF (CF (SF))` is used.

```
class SF implements Stack {
  String s = new String();
  void empty() { s = "";}
  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); };
}
class CF_on_SF extends SF
               implements Counter, Stack {
  int i = 0;
  void reset() {i = 0; };
  void inc() {i = i+1; };
  void dec() {i = i-1; };
```

```
  // lift SF to CF
  void empty() {this.reset(); super.empty();};
  void push( char a)
              {this.inc(); super.push(a) ;};
  void pop() {this.dec(); super.pop() ;};
}
class LF_on_CF_on_SF extends CF_on_SF
            implements Lock, Counter, Stack {
  boolean l = true;
  void lock() {l = false;};
  void unlock() {l = true;};
  boolean is_unlocked() {return l ;};
  // lift CF to LF
  void reset() {if (this.is_unlocked())
                   {super.reset(); }};
   void inc() {if (this.is_unlocked())
                   {super.inc(); }};
   void dec() {if (this.is_unlocked())
                   {super.dec(); }};
 // lift SF to LF
  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();}};
}
```

In this example, the above code provides for most sensible combinations, except for stack with lock only or counter with lock. In general, this translation introduces intermediate classes which may be reused for other feature combinations.

3.2. Translation via aggregation

Aggregation is a common technique for composing objects from different classes to a larger object. It is used in some object-based systems as a replacement for inheritance.

This translation requires a set of base implementations and one new class for each feature combination. The idea of the translation is to create a class where, for each selected feature, one instance variable of this type is used to delegate the services, similar to [14]. We have to be careful with delegation and calls to `this`, which should not be sent to the local object. Hence we have to supply the delegate object with the right pointer to the enclosing object, which 'replaces' `this`. For this purpose, we create a base class for each feature implementation with an extra variable which will point to the composed object. This construction enables the extension on virtual functions, which was problematic with the above translation. Here, we can check the `assumes` clauses with respect to the

full set of features, and not just with respect to the lower ones in the composition order. With the inheritance translation we had to check these assumptions for each newly added class with respect to its superclass.

Unlike the first translation, we need a few further technical assumptions. For all lifters, all methods are lifted explicitly, e.g.

```
int size() { return super.size(); };
```

is assumed to be present. Furthermore, we need to assume that instance variables which are used in lifters are declared public**. Also, the name self may not be used.

The main task of this translation is to compose the lifters, i.e. all lifters for one method have to be merged at once here. This can lead to more dense code, as all needed lifters are composed in one class, contrary to the inheritance translation.

An object creation

```
new  F1(F2...(Fn)...)
```

translates to

```
new  F1_F2_..._Fn
```

For this, we first need the following base classes for each feature implementation F_i, $i = 1 \ldots n$. For the type of self in the code below, we use the class $F_1\_F_2\_\ldots\_F_n$. If no assumes statements are used, then just I_i is sufficient and the class can be reused for other object creations. Alternatively, one can introduce an intermediate class with just the needed interfaces I_i, $I_i^1\_\ldots\_I_i^{l_i}$.

```
class Fi implements Ii {
          // reference for delegation
  (F1_F2_..._Fn) self ;

          // reference for virtual functions,
          // based on required feature list
          // constructor for this class
  Fi (F1_F2_..._Fn  s) { self = s; } ;
  Fi.vardecls
          // function implementations
  Ii.t1 θF1.f1;
  ⋮
  Ii.tki θF1.fki;
}
```

For delegation to work in the above, we need to apply a substitution θ which renames this to self:

$$\theta = [\text{this} \mapsto \text{self}]$$

With the above base implementations we construct the class $F_1\_F_2\_\ldots\_F_n$ via aggregation.

**Note that public declarations are omitted throughout this presentation.

```
class F1_F2_..._Fn implements I1, I2, ..., In {
          // delegate objects
  F1 b1 = new F1(this);
  ⋮
  Fn bn = new Fn(this);
          // now need to nest lifters
          // lift feature 2 to 1
  I2.t1 δ2F1,2.f1;
  ⋮
  I2.tk2 δ2F1,2.fk2;

          // lift feature 3 to 1
  I3.t1 δ3θ2,3F1,3.f1;
  ⋮
  I3.tk3 δ3θ2,3F1,3.fk3;
    ⋮
          // lift feature n to 1
  In.t1 δnθn−1,nθn−2,n ... θ2,nF1,n.f1;
  ⋮
  In.tkn δnθn−1,nθn−2,n ... θ2,nF1,n.fkn;
}
```

The substitutions in the above are used to unfold the appropriate lifters one after each other. We indicate the application of lifters via unfolding operators $\theta_{i,j}$, where $\theta_{i,j}$ unfolds the lifter from j to i, defined as

$$\theta_{i,j} = [\texttt{super}.f_1 \mapsto F_{i,j}.f_1, \ldots, \texttt{super}.f_{k_i} \mapsto F_{i,j}.f_{k_j}],$$

and also passes the actual parameters. Unlike in the examples below, unfolding is in general more involved for functions, as we cannot have local blocks with return statements. Hence we also assume for simplicity that methods return `void`[††].

In the combined lifters, we have to delegate calls to `super` to the delegate objects. For this purpose, δ_i shall rename the instance variables and method calls of methods in I_i to `super` correctly to the corresponding b_i. For instance, `super.pop()` is translated to `sf.pop()`, where `sf` is the name of the instance variable in the following example. We show the translation for the combination of the three introductory features. First, new base classes are introduced (with suffix `_ag`):

```
class SF_ag implements Stack {
  Stack self;
```

[††]This is no restriction, as in Java objects of primitive type can be 'wrapped' into an object in order to be passed as variable parameters.


```
  String s = new String();
  SF_ag(Stack s) {self = s;};
  void empty() { s = "";}
  void push( char a)
        {s = String.valueOf(a).concat(s);};
    // self replaces this for delegation!
  void push2( char a)
        {self.push(a); self.push(a);};
  void pop() {s = s.substring(1); };
  char top() { return (s.charAt(0)); };
}
class CF_ag implements Counter {
  Counter self;
  CF_ag (Counter s) {self = s;};
  int i = 0;
  void reset() {i = 0; };
  void inc() {i = i+1; };
  void dec() {i = i-1; };
  int size() {return i; };
}
class LF_ag implements Lock {
  Lock self;
  LF_ag (Lock s) {self = s;};
  boolean l = true;
  void lock() {l = false;};
  void unlock() {l = true;};
  boolean is_unlocked() {return l;};
}
```

A class for a composed object is shown below.

```
class LF_CF_SF implements Lock, Counter, Stack
{
           // delegate objects
  SF_ag sf = new SF_ag(this);
  CF_ag cf = new CF_ag(this);
  LF_ag lf = new LF_ag(this);
           // delegate to lock
  void lock() {lf.lock();};
  void unlock() {lf.unlock();};
  boolean is_unlocked()
    {return lf.is_unlocked();};
         // delegate to lock
  void reset()
    {if (this.is_unlocked()) {cf.reset();}};
```

```
  void inc()
    {if (this.is_unlocked()) {cf.inc();}};
  void dec()
     {if (this.is_unlocked()) {cf.dec();}};
  int  size()
     {return cf.size();};
          // delegate to stack
  void empty()
      {if (this.is_unlocked())
          {this.reset(); sf.empty();}};
  void push( char a)
        {if (this.is_unlocked())
           {this.inc(); sf.push(a);}};
  void push2( char a) {sf.push2(a);};
  void pop() {if (this.is_unlocked())
              {this.dec(); sf.pop();}};
  char top() {return sf.top();};
}
```

Compared to the first translation, we need fewer classes here, as the base classes can be reused. On the other hand, aggregation introduces another level of indirection which may affect efficiency.

4. PARAMETRIC FEATURES

In order to write reusable code, it is often desirable to parameterize a class by a type. In this section, we introduce parametric features, which are very similar to parametric classes. Due to the flexible composition concepts for features, we also need expressive type concepts for composition. For Java, parametric classes have recently been proposed and implemented in the language Pizza [8], which will be the target language for our translations. Apart from other nice extensions, which are also used in some examples here, Pizza introduces a powerful extension for type-safe parameterization. The notation for type parameters is similar to C++ templates [15]. A typical example is a stack feature parameterized by a type parameter `A`, written as `<A>`, as follows:

```
interface Stack<A> {
   void empty();
   void push( A a);
   void push2( A a);
   void pop();
   A top();
}
feature SF<A> implements Stack<A> {
           // Use Pizza's List data type
   List<A> s = List.Nil;
   void empty() { s = List.Nil;};
```

```
   void push(A a) {s = List.Cons(a,s);};
   void push2(A a)
          {this.push(a) ; this.push(a);};
   void pop() {s = s.tail();};
   A top() { return s.head();};
}
```

Stacks over type char with a counter are then created via

```
    new CF (SF<char>);
```

Note that it is sometimes useful to make assumptions on the parameter for providing operations, e.g.

```
interface Matrix<A implements Number> {
  void multiply_matrix( ...);
  ...
}
```

Such an assumption is different from assumptions via `assumes`, as it refers to a parameter and not to the inner feature combination. The difference is that this kind of parameterization is not subject to liftings.

For translating parameterization into Java we refer to [8]. We only aim at translating into Pizza. As we mostly use basic concepts, it is not necessary to go into the details of the Pizza type system. We do however use another convenient, but not essential, extension provided by Pizza: algebraic data types, which are also called class cases. With algebraic data types, we can easily define basic data structures such as lists used above. The following class declaration introduces a class of list elements which are either the empty list `Nil` or a cons node.

```
class List<A> {
   case Nil;
   case Cons(A head, List<A> tail);
```

For such a data type, an appropriate switch statement can be used, for instance for computing the length of a list:

```
int length(List<A> x) {
        switch (x) {
        case Nil: return 0;
        case Cons( A el, List<A> xs):
                  return 1 + xs.length();}
}
```

4.1. Type dependencies

For parameterized features new and interesting specification problems occur when combining features. Not only can features depend on each other, but the parameter types can also depend on each other. This gets even more complicated if more than two features are involved, as shown below. For instance,

we may want to combine `Stack<A>` with a feature which only allows elements within a certain range. Its implementation maintains two variables of type `A` used for filtering. This feature `Bound` is parameterized by a numeric type:

```
interface Bound<A> {
  boolean check_bounds(A el);
}

feature BF<A implements Number>
        implements Bound<A> {
  A min, max;
  BF(A mi, A ma) { min = mi; max = ma;};
  boolean check_bounds(A el) {...};
}
```

Clearly, we can only combine the two features when both are supplied with the same type. This can be expressed by liftings:

```
feature BF<A> lifts Stack<A> { ... }
```

Another example for such a dependence will be shown in Section 7. Note that in feature implementations, assumes conditions can also express type dependencies in the same way.

4.2. Multi-feature interactions and type interactions

In the following, we discuss multi-feature interactions and type interactions using the undo example. This will lead to a new aspect of lifting features, i.e. that lifting may change the type parameter.

The implementation idea of the undo feature is simple: save the local state of the object each time a function of the other features is applied (e.g. push, pop). Undo depends essentially on all 'inner' features, since it has to know the internal state of the composed object. As we work in a typed environment, the type of the state to be saved has to be known. This multi-feature interaction is resolved by an extra feature, called `Store`, which permits to read and write the local state of a composed object. (The motivation for store is similar to that of the Memento pattern in [16].)

We introduce the following interface for `Store`:

```
interface Store<A> {
   void put_s( A a);
   A get_s();
}
```

Note that the parameter type depends on the types of all instance variables of the used features. Consider for instance adding this feature to a stack with counter. Then for both features the local variables have to be accessed.

In spite of this problem, we can define a trivial base implementation for store. As the base implementation cannot store anything useful, we introduce a Pizza type/class `VoidT`, which has just one element, `void_el`.


```
class VoidT { case void_el; }
                    //  base implementation
feature ST implements Store<VoidT> {
  void  put_s(VoidT a ) {};
  VoidT get_s() {return VoidT.void_el; };
}
```

With the `Store` feature, we reduce the multi-feature interaction to a type interaction problem. This means that the parameter type of a feature has to change when a feature is lifted. The following solution makes these type dependencies explicit. We use polymorphic pairs via the Pizza class `Pair<A, B>`, defined as

```
class Pair<A,B> {
    public case Pair(A fst, B snd);
    ...
}
```

This class is useful for type composition. In the lifter below, we state that the inner feature combination supports feature `Store<A>` for some type `A`. For this, we need a new syntactic construct, namely `assumes inner`. As feature stack `ST` adds an instance variable of type `List<B>`, we can support the store feature with parameter `Pair<List<B>,A>`.

```
feature ST<B> lifts Store<Pair<List<B>,A>>
              assumes inner Store<A> {
   Pair<List<B>,A> get_s()
      {return Pair.Pair( s,     // local state
          super.get_s()); }     // inner state
   void get_s(Pair<List<B>,A> s) { ... }
}
```

The `assumes inner` construct has some constraints. The lifted feature may not have instance variables or virtual calls to self where the type parameter is used which is changed in the lifter (here `A`). (This can be allowed if the type change is a specialization, which this is not the case in this example.)

This inner condition is implicit in other lifters and is only needed if the type parameters change. The lifting

```
feature F lifts F1<A> { ... }
```

can be seen as an abbreviation for

```
feature F lifts F1<A>
          assumes inner F1<A> { ... }
```

We show below how this change of parameters affects the two translation schemes of Section 3. Continuing with the example, we express that the counter `CF` adds an integer and `LF` adds a Boolean variable with the following lifters:

```
feature CF lifts Store<Pair<A, int>>
           assumes inner Store<A> {
   ...
}
feature LF lifts Store<Pair<A, Boolean>>
           assumes inner Store<A> {
  ...
}
```

With the above lifters, we can assure that the store feature works correctly and with the correct type for any feature combination. We can now write the generic undo feature, which can be plugged into any other feature combination. It is important that the store feature fixes the type of the state of the composed object. The undo feature can then have an instance variable of this type. Recall that this is not possible for store, as the type parameter of store changes under liftings.

The undo feature consists of two parts: storing the state before every change and retrieving it upon an undo call. The latter is the core functionality of undo, whereas the former will be fixed for each function which affects the state via liftings. First consider the undo feature and its implementation, which uses a variable `backup` to store the old state. Since there may not be an old state, we use the algebraic (Pizza) type `Option<A>`, which contains the elements `None` or `Some(a)` for all elements `a` of type `A`.

```
interface Undo<A> {
   void undo();
}
class Option<A> {
     case None;
     case Some(A value);
}
feature UF<A> implements Undo<A>
              assumes Store<A> {
  Option<A>  backup = None;
  void undo() {
        switch ((Option) backup) {
           case Some(A a):
              put_s( a );
} } }
```

An alternative version of undo may store several or all old states. Due to our flexible setup, we can just exchange such variations.

For each of the other features, we have to lift all functions which update the internal state. As for lock, this lifting is canonical, e.g. for `push`:

```
  void push(A a) {
     backup = Option.Some( get_s());
     super.push(a); };
```


Note that there is an interesting interaction between lock and undo: shall undo reverse the locking or shall lock disable undo as well? We chose the latter for simplicity and hence add lock after undo. Lifting undo to lock is canonical and not shown here. As an example, we can create an integer stack with undo and lock as follows:

```
new LF (UF<Pair<int,VoidT>> (SF<int>(ST)))
```

4.3. Translation into Pizza

We show in the following how to translate the above extensions into Pizza. This will reveal another difference between aggregation and inheritance: for inheritance, we cannot cope with the change of parameters. Otherwise, the translation to Pizza is quite simple.

For aggregation, additional inner statements just translate into types of the instance variables of the class generated for a combination. This is shown in the following code for a class generated for a composed object with both stack and store features. We first introduce a class `SF_ag<A>` for parametric stacks. As we do not allow calls to self for features whose type parameter changes during lifting, we do not use the usual delegation mechanism in the above class. Hence we use just `ST`. The class `SF_ST<A>` exports the interface `Store<Pair<List<A>,VoidT>>`, but uses a delegate object with interface `Store<VoidT>`.

```
class SF_ag<A> implements Stack<A> {
   Stack<A> self ;
   SF_ag(Stack<A> s) {self = s;};
   List<A> s = List.Nil;
   void empty() ...
}
class SF_ST<A> implements Stack<A>,
                   Store<Pair<List<A>,VoidT>>{
   SF_ag<A>     sf = new SF_ag(this);
   Store<VoidT> st = new ST();
   Pair<List<A>,VoidT> get_s()
        {return Pair.Pair(sf.s, st.get_s());};
   ...
}
```

A further detail to observe is that all type variables have to be considered for the translation. This means that for the new class introduced, all type variables which appear as parameters in the desired set of features have to appear as parameters. For instance, for the combination `F<A>(G<B>)`, we need a class `F_G<A,B>`.

For inheritance, an inner statement is an assumption on the extended class. For instance, we cannot translate the above feature combination to the following (illegal) code:

```
        // illegal ! Type conflict!
class ST_SF<B,A> extends ST<A>
     implements Stack<B>,
```

```
                Store<Pair<List<B>,A>> {
    Pair<List<B>,A> get_s() {
        Pair.Pair( s,           // local state
               super.get_s()); // inner state
   }
   ...
}
```

This (in Pizza illegal) code attempts to extend a class `ST<A>` which implements `Store<A>` by a class implementing `Store<Pair<List<B>,A>>`. If the parameter changes, this amounts to specialization for parameterized classes, which is problematic in typed imperative languages, as discussed in [8]. In Pizza, subtyping does not extend through constructors such as `Pair`, hence `Store<Pair<List<B>,A>>` is not a subtype of `Store<A>`. If the parameters do not change, the translation is straightforward.

5. GENERIC LIFTINGS VIA HIGHER-ORDER FUNCTIONS

For some features, the lifters are schematic and do not depend on the lifted feature. For instance, in the stacks example it is easy to see that lifting to lock just adds a check, regardless of the lifted feature. It is natural to express this by higher-order functions. We call this generic liftings, which are implemented via higher-order functions in Pizza. Pizza function types can be polymorphic as well and are, in general, written as `(A1, A1, ..., An)->A0`, where `Ai` are types. Consider for instance the following function for lifting lock, where `()->void` is the Pizza notation for a function type.

```
void lift_to_lock( ()->void f)
       { if (this.is_unlocked() ) { f() ;};}
```

It can be used e.g. with a lifter

```
void reset() {lift_to_lock(super.reset);};
```

which replace repeated code in lifters, e.g. in

```
void reset()
  {if (this.is_unlocked()) {super.reset();}};
```

It can be useful to extend the language to allow default lifters which are applied if no explicit lifters are provided.

Note that there is a small problem with generic lifters in Java: we cannot write a generic lifter which suits both functions and procedures (without return values) in Java. This is of course only useful in cases where the return value of the lifter is generic as well. Consider for instance the above lifter: it cannot be applied to lift the function pop. A lifter for this case would look like:

```
void lift_fun_to_lock( ()->char f)
           {return  f();}
```

This code may replace the lifters for `top` and `size` to Lock. (Note that disabling these two functions via lock is more difficult. What shall these lifted functions return? The only two options are raising an exception instead or returning an arbitrary or default value.)

6. FEATURES AND EXCEPTIONS

So far, we have used lifters to adapt features to the context of other features, which usually meant additional state in the form of instance variables. Yet features may equally well add exceptions. Consider for instance the stack example and adding exception handling for stack underflow. The idea is to add a feature which does nothing but raising the appropriate exception. This feature can be added or omitted as desired.

```
  // exception for underflow
class Underflow extends Exception {
}
interface UnderflowI {
   void display_exception();
}
feature Underflow implements UnderflowI {
  void display_exception()
    {System.out.println("Stack Underflow!");};
}
```

In the above example, the underflow feature only implements some auxiliary functions, for instance the function `display_exception`. (For a different example see Section 7.3.) Exceptions can be raised in the lifters:

```
feature Underflow lifts Stack {
  public void pop() throws Underflow
   { if (is_empty()) throw new Underflow();
                     else super.pop(); }
}
```

This example shows that the feature model easily accommodates exceptions, but in Java a small language extension can be useful. 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, but all possibly affected methods must be declared as raising the exception, regardless if the exception feature is used.

For instance, in the above example, the underflow exception has to be declared for the `pop` function at every occurrence. As we can use lifters to add exceptions, this seems too strict. Hence we argue that only the lifters should declare exceptions in this case. The translation to Java in turn has to add declarations of exceptions to function definitions appropriately. This may require creating a class twice, with and without exception handling. As the main idea is straightforward, we refrain from presenting this in detail.

7. EXAMPLES

In the following sections we sketch a few typical applications for feature-based programming. These range from generic application frameworks to application domains where change is predominant and flexibility is required.

7.1. Variations of design patterns

We show in the following that for several typical programming schemata—coined design patterns [16]—feature-based implementations provide high flexibility and the desired reusability. This is particularly important if several features or design patterns are combined. The following examples are freely taken from standard literature on design patterns [16].

7.1.1. Proxy pattern

Consider implementing some functional entity, e.g. sets, where caching of the results of operations is a viable option. In the lines of [16], this can be viewed as a Proxy pattern. Clearly, a cache is an independent feature, and there exist many variations of caching, e.g. considering the data structures used or the replacement strategy. Furthermore, it may depend on appropriate hash functions, which could also be provided via features.

When writing a reusable set of caching modules, the various cache implementations just implement the data structures and the access functions. Interaction resolution in turn modifies the access operations for the object to be cached and determines the type dependencies.

Consider writing this with object-oriented languages: for each needed combination of a cached object, a cache, and a hashing function, a new (sub-)class has to be implemented.

A sketch of such an example is shown below. It shows how to add a cache to the parametric features `Set<A>` and `Dictionary<A, B>`. For caching, these data structures are viewed as mappings, from `A` to boolean and `A` to `B`, respectively. The feature implementation `CacheI<A,B>` (whose interface `Cache` is not shown here) caches mappings from `A` to `B`. Note that the lifters express the type dependencies.

```
interface Set<A> {
   void put( A a);
   boolean contains(A a);
}
interface Dictionary<A, B> {
  Option<B> get(A key);
  void put(A key, B value);
}
feature CacheI<A,B> implements Cache<A,B> {
   ...
   void put_s( A a, B b) {...} ;
   boolean find_s(A a) {...};
   B get_s() {...};
}
feature CacheI<A,B> lifts Dictionary<A, B> {
     // adapt access functions to cache
  Option<B> get(A key)
    {if find(key) return Option.Some(get_s());
            else  return super.get();  }
```

```
  ...
}
     // second parameter is just boolean here
feature CacheI<A,boolean> lifts Set<A> {
  ...
}
```

7.1.2. Adaptor patterns

The adaptor design pattern [16] glues two incompatible modules together. This design fits nicely in our setting, as adaptors should be reusable. Typically, there is some core adaption functionality, e.g. some data conversion, which we model as a feature. When adding this to another feature, we can just lift the incompatible functions with the help of the core functionality.

An example is converting big endian encoding of data to little endian. For instance, if we output data on a (low-level) interface which needs big endian, but we work with little endian, such a conversion feature can just be added. The adaptor feature provides the core functionality, here the data conversion, and interaction resolution adapts the operations of the object.

The following features and lifters sketch the solution of pluggable adaptors with features. The adaptor feature `Big_to_little_endian` adds a conversion function, which is used in the lifter to provide the put method with big endian data input.

```
feature Big_to_little_endian {
            // convert to little endian
  int big_to_little(int a) {...};
}
             // assumes little_endian
interface low_level_IO {
  void put(int a);
}
feature Big_to_little_endian
        lifts low_level_IO {
 void put(int a)
     {super.put( big_to_little(a)); };
}
```

There is an interesting optimization if the cache feature is used with the above adaptor. The interaction handling for the two feature as it is not needed to convert the cached data to big endian.

7.2. Automata-based modeling techniques

Diagrammatic modeling techniques have gained large interest in the last years. We discuss here automata-based description techniques, where we aim at composing hierarchical automata from smaller automata parts. In particular, we discuss interactions arising in such descriptions. We follow an example which was studied in [17,18] 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 [17], there arise

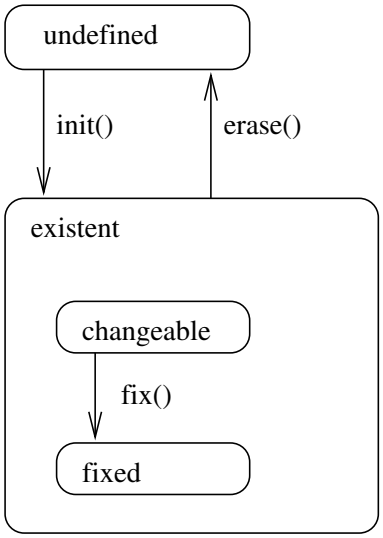


Figure 4. Hierarchical automaton for the status of a document.

feature interactions, which are not easily modeled by simple automata concepts. We show here how to model automata and the interactions by feature-oriented programming. The main benefit we gain is composition of automata, where our model of feature interactions applies. This may provide 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 feature-oriented programming. This flexible composition is the main difference to the many other techniques for graphic description of software components.

In this example, documents are manipulated via a groupware application which admits several options or features for controlling the access to the documents. These features can be selected individually for each document. They usually model the current state of each document to control the allowed operations. We model the following, very basic features as automata:

- 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 4. 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 assumes clause. Feature `BA` allows `DA`, the implementation of DocAut, to use some basic functionality of `BA` to construct automata. (This allows one 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 assumes 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 assumes 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 assumes 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 [18]. Note that the 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.

We have shown that typical composition problems of automata descriptions can be modeled by our feature model. 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 [17,19].

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 [19] and in Hierastates [18]. 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 [19]. 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 explicitly by lifters. It remains for future work to extract a broad set of typical interactions and to devise graphical notation for them.

7.3. Feature interaction in telecommunications

A well known feature interaction problem 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 [20].

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

- forwarding of calls;
- ICS (incoming call screening); and
- 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.

7.3.1. 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, which is 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.

```
class Ph implements Phone {
    // origin of call
  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; } }
}
```

7.3.2. 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, for simplicity, not limited.)

7.3.3. 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 assumes Phone{
  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));};
}
```

7.3.4. 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 with respect to the next hop. If only a check with respect to the origin of the call is desired, one just has to adapt the lifter (and not to update the origin of the call). Thus, lifting allows for a modular resolution of the interaction between two features.

7.3.5. 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. We only show the interface here for brevity:

```
interface OcsI {
   int ocs(int dest) throws Busy;
}
```

7.3.6. 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:

```
   // returns 6
 println( con.connect(5));
   // returns 6
 println( con.connect(6));
   // returns 8, as forwarded to 8,
   // which is allowed by ICS
 println( con.connect(7));
   // returns Busy, as forwarded to 8,
      which is not allowed by ICS
 println( con.connect(8));
   // returns 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 one 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.

8. DISCUSSION

We discuss in the following the main assumptions and characteristics of our feature composition approach, followed by an overview of related work and some extensions. Our feature model enables the flexible composition of features and lifters, which can greatly reduce complexity. Our main claim is that there is a large class of applications for our expressive feature composition method. Recall that our approach generalizes inheritance and aggregation and integrates nicely with several common language extension.

In general, feature interactions are semantic problems of two or more features (and not a priori of their implementations). Our composition method for features is designed to clarify dependencies and to provide structure and flexibility. In this sense, we do not address the general (semantic) feature composition problem but discuss programming-level composition of feature implementations. No general-purpose technique, including ours, can solve the (usually undecidable) problem of detecting semantic feature interactions.

A premise of our approach is that any interactions can be handled for two features at a time. This means that interactions between three or more features cannot be modeled directly if they do not show up between two individual features. In our experience, such cases do not appear frequently in practice. For instance, in the literature on feature interactions in telecomunications, the survey in [20] reports 22 feature interactions, but none of them with three features. (And hence none where the interactions do not show up when considering two features at a time.) When true multi-feature interactions examples have been identified (which do not indicate an inconsistency), there are two options. If the features are highly entangled, it is preferable to write code for particular combinations. Otherwise, there are usually ways to circumvent the interaction or reduce it to two-feature interactions. For instance, a feature can be split into two separate features. Recall for instance the undo feature which assumes the store feature to be added before (with respect to the feature order) the features to be undone.

Consider the following example for a true three-feature interaction. Assume an email client with added features for handling priorities of emails (bulk, normal, priority) as follows.

1. Basic email client; send and receive emails.
2. Downclass outgoing mails (at least for some users).
3. Discard incoming bulk mail.
4. Forward incoming emails to another address.

With the assumptions below we construct a true three-feature interaction.

- The basic client does not interpret priorities.
- Downclassing is destructive on the email (no copying).
- Forwarding takes place before delivery of incoming mail.
- Features are composed in the above order. (Which is drawn upside down in earlier pictures.)

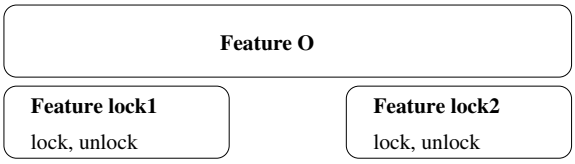


Figure 5. Alternative feature composition.

With all the above features, an incoming normal message may be forwarded, and hence downclassed to bulk by the downclass feature. As the email is modified to bulk, it is discarded by the incoming bulk filter. In brief, the email is forwarded but never delivered to the client. This undesired interaction does not happen with two features at a time.

Yet, as in most other cases, it is easy to avoid (e.g., by reordering features or different implementations). This may incur implementation and runtime overhead in some cases, which is the price for flexibility and robustness.

Compared to the above example, a case which occurs more frequently is when performance optimizations are possible only for particular combinations. For instance, for the combination of the stack, counter, and lock feature, in a call to `push` the locking of the stack is checked twice. Clearly, this small performance loss can be avoided. Some simpler optimizations can be performed by a compiler. For complex ones, one would need a domain-specific optimizer, as for instance pursued by aspect-oriented programming [21].

Another major assumption of our approach is that we compose features following an ordering between features. This is given by the feature lifters. Only from the outside interface it is possible to view an object as composed of a set of features. There are several reasons for this ordering. First, it is in the spirit of inheritance and it seems to be the simplest structure capturing the essential object-oriented ideas like inheritance. Second, there are problems when viewing features as unordered. Clearly, when features are fully independent, it is not needed to order them. A simple syntactic extension may alleviate the problem.

Although one can imagine more complex feature composition schemes, we argue that basic language feature should be as lean as possible. This favors efficient implementations and clarity to the programmer. We show next that composition schemes without order have difficulties with respect to liftings or inheritance. The problem seems to be similar to known problems with multiple inheritance.

Consider an example of an object integrating two unordered features, both implementing a lock. Such a configuration with lock1 and lock2, to which a feature O is added, is shown in Figure 5. The interaction is that closing lock1 should also close lock2 and vice versa. Hence we need liftings from the two features to feature O. The simple lifting model is to lift the functions of each feature to O, e.g. by applying all lifters corresponding to the other present features. The lifter of lock1 shall call `lock2.lock();` and similarly the lifter for lock2 calls `lock1.lock();`. The problem is which version of lock should be called, the lifted or the original of the feature? If the original one is called, then all other liftings are ignored, e.g. if other features are involved. Or if the lifted version is called, then the procedure diverges.

An alternative solution would be to introduce a 'global' lock function and to lift each local lock to the global one. Then a flexible number of locks can be handled by one access point. This is another example where introducing extra features (and ordering) is useful and leads to a less ambigous structure.

8.1. Related work

We briefly compare the feature model to other approaches. Our model generalizes several of these, e.g. mixins, before and after messages in Lisp and composition filters.

- Mixins [2] have been proposed as a basic concept for modeling other inheritance concepts. The main difference is that we consider interactions and separate a feature from interaction handling. If mixins are used also as lifters, then the composition of the features and their quadratic number of lifters has to be done manually in the appropriate order.
- Method combination with before, after and around messages in CLOS [3] follows a similar idea as interactions. As with mixins, this does not consider interactions between two classes/features and gives no architecture for composition of abstract subclasses. Such after or before messages can be viewed as a particular class of interactions.
- Composition filters have been proposed in [22] to compose objects in layers, similar to the feature order in our approach. Messages are handled from outside in by each layer. The main difference is that we consider interactions on an individual basis and separate a feature from interaction handling.
- Merging different aspects, functional and non-functional, of a software system is pursued by aspect oriented programming [21]. This can be viewed as an even more general goal than feature composition, but is so far pursued only for certain application domains. We hope that feature-oriented programming can contribute to a more general theory of aspect composition.
- Several other approaches allow to change class membership dynamically or propose other compositions mechanisms [23–27]. Note that one of the main ingredients for feature-oriented programming, lifting to a context, can also be found in [26]. All of these do not consider a composition architecture as done here, and address other problems, such as name conflicts. Clearly, the idea of features can also be applied to dynamic composition, but this remains for future work. Due to dynamic change, static typing can be problematic.
- Subject-oriented programming [28] has been proposed as a model for capturing different views or roles of the same object. Feature-oriented programming can contribute to this idea as a composition technique, as different views may clearly interact.
- Another approach to flexible software libraries using code generators was presented in [6,29]. This approach also uses layers for composition, but does not consider interactions between components.

Compared to many other extensions of inheritance, the feature model contributes the following ideas:

- The core functionality is separated from the interaction resolution.
- It allows to create objects (or classes) freely by composing features.
- For the composition, we provide a composition architecture, which generalizes inheritance.

Inheritance has often been criticized as breaking encapsulation. We cannot fully relieve this problem, as liftings still break inheritance, but at least this part of the code is separated from the core feature.

8.2. Extensions

We discuss in the following a few extensions and issues which have not been addressed so far. As we have focused on feature composition, several interesting aspects have been neglected.

- We have not discussed visibility of attributes of features here, as this does not belong to our core ideas. Currently, features can use only the interfaces of the assumed features, whereas the instance variables of concrete other features are not visible (for good reason). Lifters adapt a feature interface over a concrete feature constructor. Hence only the instance variables of the concrete feature are visible.
  Hiding some functions is clearly needed in some applications. For instance, when adding the counter to a stack, we may not want to inherit the inc and dec functions, as they may turn the object into an inconsistent state.
- In this presentation, liftings depend on one concrete object constructor and an interface, which can be impractical if the interaction resolution depends on the concrete constructor which provides the methods of the interface. It is easy to imagine an extension for this purpose, but it is currently unclear if such a violation of encapsulation is needed.
- An extension to distributed objects should also address the problem of inheritance anomaly [30,31], as this addresses similar interaction problems.
- Instead of letting features assume other features, it can also be useful for specification purposes to disallow features, similar to canceling other mixins in [1].

9. CONCLUSIONS

Feature-oriented programming is an extension of the object-oriented programming paradigm. Whereas object-oriented programming supports incremental development by subclassing, feature-oriented programming enables compositional programming. Overriding as in inheritance is generalized to resolving feature interactions.

The recent interest in feature interactions, mostly stemming from multimedia applications [12,13], shows that there is a large demand for expressive composition concepts where objects with individual services can be created. It also shows that our viewpoint of inheritance as interaction is a very natural concept.

Compared to classical object-oriented programming, feature-oriented programming provides much higher modularity and flexibility. Reusability is simplified, since for each feature, the functional core and the interactions are separated. This difference encourages to write independent, reusable features and to make the dependencies to other features clear. In contrast, inheritance with overwriting mixes both, which often leads to highly entangled (sub-)classes.

ACKNOWLEDGEMENTS

The author is indebted to the reviewers for their helpful comments which helped to improve the paper.

REFERENCES

1. Steyaert P, Codenie W, D'Hondt T, De Hondt K, Lucas C, Van Limberghen M. Nested mixin-methods in Agora. *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, Kaiserslautern, Germany, July 1993 (*Springer-Verlag Lecture Notes in Computer Science*, vol. 707), Nierstrasz O (ed.). Springer: Berlin, 1993.
2. Bracha G, Cook W. Mixin-based inheritance. *Proceedings of OOPSLA ECOOP '90* 1990; *ACM SIGPLAN Notices* **25**(10):303–311. Meyrowitz N (ed.).
3. Lawless JA, Molly M. *Understanding CLOS: The Common LISP Object System*. Digital Press: Nashua, NH, 1991.
4. Opdyke WF, Johnson RJ. Refactoring: An aid in designing application frameworks. *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications*. ACM-SIGPLAN, September 1990.
5. Gosling J, Joy B, Steele G. *The Java Language Specification*. Addison-Wesley: Reading, 1996.
6. Batory D, Singhal V, Sirkin M, Thomas J. Scalable software libraries. *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, December 1993; 191–199.
7. Stein LA. Delegation is inheritance. *ACM SIGPLAN Notices* 1987; **22**(12):138–146.
8. Odersky M, Wadler P. Pizza into Java: Translating theory into practice. *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.
9. Prehofer C. From inheritance to feature interaction. *Special Issues in Object-Oriented Programming. ECOOP 1996 Workshop on Composability Issues in Object-Orientation*, Heidelberg, 1997. Mühlhäuser M *et al.* (eds.). dpunkt-Verlag, 1997.
10. Prehofer C. From inheritance to feature interaction or composing monads. *Arbeitstagung Programmiersprachen, Tagungsband der GI-Jahrestagung*. Springer-Verlag: Berlin, 1997. Also appeared as *Technical Report TUM-I9715*, Technical University München.
11. Moggi E. Notions of computation and monads. *Information and Computation* 1991; **93**(1).
12. Zave P. Feature interactions and formal specifications in telecommunications. *IEEE Computer* 1993; **26**(8).
13. Cheng KE, Ohta T (eds.). *Feature Interactions in Telecommunications III*. IOS Press: Tokyo, Japan, 1995.
14. Johnson RE, Zweig JM. Delegation in C++. *Journal of Object-Oriented Programming* 1991; **4**(3).
15. Stroustrup B. *The C++ Programming Language* (2nd edn). Addison-Wesley: Reading, 1991.
16. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design*. Addison-Wesley: Reading, MA, 1994.
17. Teege G. Hierastates: Flexible interaction with objects. *Technical Report TUM-I9441*, TU München, 1994.
18. Teege G. Hierastates: Supporting workflows which include schematic and ad-hoc aspects. *Proceedings of 1st International Conference on Practical Aspects of Knowledge Management PAKM'96*, Wolf M, Reimer U (eds.), Basel 1996.
19. Harel D, Naamad A. The statemate semantics of statecharts. *IEEE Transactions on Software Engineering Method* 1996.
20. Cameron EJ, Griffeth N, Linand Y-J, Nilson ME, Schnure WK, Velthuijsen H. A feature interaction benchmark for in and beyond. *Feature Interactions in Telecommunications Systems*, Bouma LG, Velthuijsen H (eds.). IOS Press: Amsterdam, 1994; 1–23.
21. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J. Aspect-oriented programming. *ECOOP '97* (*Springer-Verlag Lecture Notes in Computer Science*, vol. 1241). Springer: Berlin, 1997.
22. Bergmans L, Akşit M. Composing synchronization and real-time constraints. *Journal of Parallel and Distributed Computing* 1996; **36**(1):32–52.
23. Lehrmann Madsen O, Moller-Pedersen B, Nygaard K. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley: Reading, MA, 1993.
24. Ungar D, Smith RB. Self: The power of simplicity. *Lisp and Symbolic Computation* 1991; **3**(3).
25. Fröhlich HJ. Prototype of a run-time adaptable object-oriented system. *PSI '96 (Perspectives of System Informatics)*, Akademgorodok, 1996 (*Springer-Verlag Lecture Notes in Computer Science*, vol. 1755). Springer: Berlin, 1996.
26. Seiter LM, Palsberg J, Lieberherr KJ. Evolution of object behavior using context relations. *IEEE Transactions onSoftware Engineering* 1998; **24**(1).
27. Mezini M. Dynamic object modification without name collisions. *ECOOP '97* (*Springer-Verlag Lecture Notes in Computer Science*, vol. 1241). Springer: Berlin, 1997.
28. Harrison W, Ossher H. Subject-oriented programming (a critique of pure objects). *Proceedings of OOPSLA 1993* 1993; *ACM SIGPLAN Notices* **28**:411–428. Paepcke A (ed.). ACM Press.
29. Batory D, Geraci BJ. Composition validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering* 1997; **23**(2).
30. Matsuoka S, Yonezawa A. Analysis of inheritance anomaly in object-oriented concurrent programming languages. *Research Directions in Concurrent Object-Oriented Programming*, Wegner P, Agha G, Yonezawa A (eds.). MIT Press, 1993; 107–150.
31. Lechner U, Lengauer C, Nickl F, Wirsing M. (Objects + concurrency) & reusability—A proposal to circumvent the inheritance anomaly. *Proceedings ECOOP '96*, Linz, Austria, July 1996 (*Springer-Verlag Lecture Notes in Computer Science*, vol. 1098), Cointe P (ed.). Springer: Berlin, 1996; 232–247.

