Feature-Oriented Programming: A Fresh Look at Objects

Christian Prehofer

Institut für Informatik, Technische Universität München,
80290 München, Germany, prehofer@informatik.tu-muenchen.de

Abstract. We propose a new model for flexible composition of objects from a set of features. Features are similar to (abstract) subclasses, but only provide the core functionality of a (sub)class. Overwriting other methods is viewed as resolving feature interactions and is specified separately for two features at a time. This programming model allows to compose features (almost) freely in a way which generalizes inheritance and aggregation. For a set of n features, an exponential number of different feature combinations is possible, assuming 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 further discuss parameterized features, which work nicely with our feature model and can be translated into Pizza, an extension of Java.

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 approaches to increase flexibility (mix-ins [18, 2], around-messages in Lisp [8], class refactoring methods [12]) and to approaches using different composition techniques, such as aggregation and (abstract) subclasses.

In this paper we propose a new model for object-oriented programming which nicely generalizes inheritance and includes the above mentioned extensions and new 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]. The main difference is that we separate the core functionality of a subclass from overwriting methods of the superclass. We view overwriting 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. Similar to inheritance, this is accomplished by method overwriting, but lifters depend on two features and are separate entities used for composition. In contrast, inheritance just overwrites methods of the superclass.

Our new model allows to compose objects from individual features (or abstract subclasses) in a fully flexible and modular way. Its main advantage is that

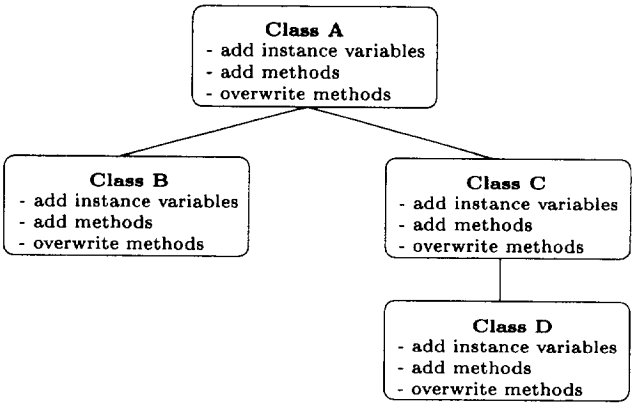

Fig. 1. Typical Class Hierarchies

objects with individual services can be created just by selecting the desired features, unlike object-oriented programming. Hence feature-oriented programming is particularly useful in applications where a large variety of similar objects is needed. 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 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 overwriting) which is similar to constructing a concrete class hierarchy. To construct an object, features are added one after another in a particular order. (As we only compose objects, there is no real notion

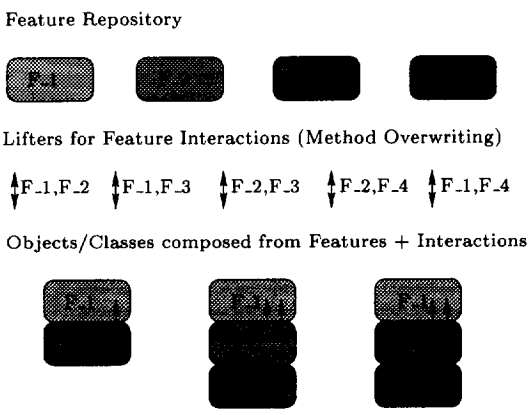

Fig. 2. Composing Objects in the Feature Model

of a class, which is hence often confused with the (type of) objects.) 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 of two features at a time, there is only a quadratic number $\binom{n}{2} = \frac{n^2-n}{2}$ 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 5 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). The observation that most, but not all, interactions can be handled for two features at a time is a major premise of this approach.

We show that feature-oriented programming generalizes object-oriented techniques and gives a new conceptual model of objects and object composition. To support this, we will show how to create Java [6] code for concrete feature selections, first using inheritance and then using aggregation and delegation. This shows the relations with known techniques and compares both techniques. In fact, we will show two cases where aggregation is more expressive than inheritance, refining earlier results [17].

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, as in many cases subclasses should be an independent entity, and not a subclass. This also makes class refac-

toring [12] much easier and sometimes unnecessary. The idea is similar to abstract classes, but we also cover dependencies between features.
- 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.
- As we consider only liftings or interactions between two features at a time, the model is as simple as possible. In case of dependencies between several features, liftings between two features can still suffice, if we consider auxiliary features (see Sec. 4.2).

The technical contributions and results in this paper are as follows:

- Translations of a feature-based language extension of Java into 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 [11], 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.

The origin of this idea of features in fact goes back to applications of monad theory in functional programming, as discussed in [13, 14]. In this earlier paper, composition of state monads was compared to inheritance and extended to other monads in functional programming. The motivation for this work was the recent development in telecommunication and multimedia software, where feature interactions have recently attracted great attention [21, 3]. Examples for feature-oriented programming in this area are discussed in [13, 15].

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. Examples in Section 5 and discussions of the approach in Section 6 and Section 7 conclude the paper.

2 A First Example for Feature-Oriented Programming

In this section, we introduce feature-oriented programming with 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.

Note that we only treat stacks over characters; parametric stacks will be considered later. We first define interfaces for features. Although not strictly needed for our ideas, they are useful if there are several implementations for one interface. Furthermore, they ease translation into Java, as a class can implement several interfaces in Java.

```
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();
}
```

The code below provides base implementations of the individual features. 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 subclasses. Such lifters are separate entities and always handle two features at a time. In the following code, features (via 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 (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();}};
}
```

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 7.

The modular specification of the three 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 combinations. The resulting objects behave as desired. In addition, we can of course 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

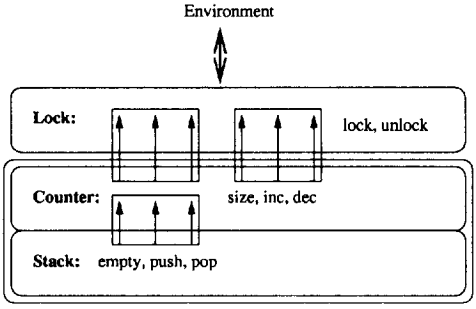

Fig. 3. Composing features (rounded boxes) by lifters (boxes with arrows)

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.

The composed object 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 run 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. in the following example:

```
feature DisplayAdapter assumes AsciiPrintable  {
  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 usual 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 aggrega-

tion with delegation. Hence this also serves to compare the feature model with both of these approaches and will highlight two cases where both differ.

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
- $F_{i,j}.f_{k_j}$ code for lifting $I_j.t_{k_j}$

```
interface I_1 {
    I_1.t_1;                              // method declarations
    ⋮
    I_1.t_{k_1};
}
    ⋮
interface I_m {
    I_m.t_1;
    ⋮
    I_m.t_{k_m};
}
feature F_1 implements I_1 assumes I_1^{l_1}, ..., I_1^{l_n} {
    F_1.vardecls                          // variable declarations
    I_1.t_1 F_1.f_1;                      // 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};
}
```

For this schematic program, concrete object creations can be translated into Java in two ways, reflecting two object-oriented programming techniques: aggregation and inheritance. For both 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 the following:

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 $this.fct(\ldots)$ instead of $fct(\ldots)$.
3. Variable declarations have no initializations.

3.1 Translation via Inheritance

For this translation, we create a concrete set of classes, one extending the other, for each used feature combination $F_1(F_2(F_3(\ldots(F_n)\ldots)))$. First a new class $F_1\_F_2\_F_3\_\ldots$ is introduced, which extends $F_2\_F_3\_\ldots$, followed by a class $F_3\_\ldots$. The class $F_1\_F_2\_F_3\_\ldots$ adds the functionality for interface I_1 and lifters for all others.

Formally, an object creation

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

translates to

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

Furthermore, we need the following Java classes for $i = 1, \ldots, n$:

```
class F_i_F_{i+1}_..._F_n extends F_{i+1}_..._F_{n-1} 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};
}
```

Observe that $n - i$ lifters are needed, which may call methods of the super class. The translation assumes that the features required for F_i via `assumes` are present in the extended class. Otherwise, undeclared identifies occur in the translated code, which would only be allowed in a dynamically typed language. 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, our 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 [7]. 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 us to check the `assumes` clauses globally, i.e. wrt the newly created set of features. With the inheritance translation we had to check these assumptions for each newly added class wrt 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.[1] 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` $F_1(F_2 \ldots (F_n) \ldots)$

translates to

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

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 F_i implements I_i {
    (F_1_F_2_..._F_n) self ;                 // reference for delegation
    F_i (F_1_F_2_..._F_n s) { self = s; } ;  // constructor for this class
    F_i.vardecls
    I_i.t_1 θF_1.f_1;                        // function implementations
    ⋮
    I_i.t_{k_i} θF_1.f_{k_i};
}
```

[1] Note that public declarations are omitted throughout this presentation.

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.

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

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

For simplicity, we only indicate the applications of nested lifters via unfolding operators θ_i, where $\theta_{i,j}$ unfolds the lifter from j to i, sketched 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`.[2]

Furthermore, 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`):

[2] 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.

```
class SF_ag implements Stack {
  Stack self;
  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 proper 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 just recently been proposed and implemented in the language Pizza [11], 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 rather powerful extension for type safe parameterization. The notation for type parameters is similar to C++ templates [19]. A typical example is a stack feature parameterized by a type `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> {
   List<A> s = List.Nil;        // Use Pizza's List data type
   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 [11]. We here 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.

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 dependency will be shown in Section 5. 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 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 solved by an extra feature, called `Store`, which allows to read and write the local state of a composed object. (The motivation for store is similar to the Memento pattern in [5].)

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.

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 the Pizza class `Pair<A, B>`, providing for polymorphic pairs, for type composition. In the following lifter, 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` however has some constraints. The lifted feature may not have instance variables or calls to self where the changed type parameter type is used. (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 translations schemes of Section 3. Continuing with the example, we express that the counter `CF` adds an integer and `LF` 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. All we need to add is a base implementation for store. As the base implementation cannot store anything useful, we introduce a Pizza type/class `Void`, which has just one element, `void_el`.

```
class Void { case void_el; }

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

With the store feature, 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 with 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,Void>> (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 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>,Void>>`, but uses a delegate object with interface `Store<Void>`.

```
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>,Void>>{
   SF_ag<A>    sf = new SF_ag(this);
   Store<Void> st = new ST();
   Pair<List<A>,Void> 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. If the parameter changes, this amounts to specialization for parameterized classes, which is problematic in typed imperative languages, as discussed in [11]. In Pizza, the problem in this example is that subtyping does extend through constructors such as `List`. 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
   }
   ...
}
```

If the parameters do not change, the translation is straightforward.

5 Examples

In the following, we sketch a few more typical applications for feature-based programming. Some examples are freely taken from standard literature on design patterns [5]. We argue that for many of these typical programming schemata, feature-based implementations provide high flexibility and the desired reusability. This is particularly important if several features or design patterns are combined.

5.1 Adding a Cache

Consider implementing some functional entity, e.g. sets, where caching of the results of operations is a viable option. In the lines of [5], this can be viewed as a Proxy pattern. Clearly, a cache is an independent feature, and there exist many variations of caching. For instance, considering the data structures used and the replacement strategy. And it furthermore 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 classical 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>`. The feature implementation `CacheI<A,B>` (whose interface `Cache` is not shown here) caches mappings from `A` to `B`.

```
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> {
  ...
}
```

Note that the lifters express the type dependencies. For instance the set is viewed as a mapping from `A` to `boolean`.

5.2 Adaptor Patterns

The adaptor design pattern [5] 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 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,

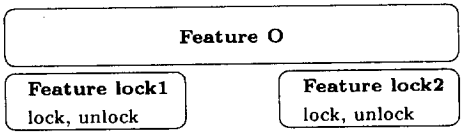

Fig. 4. Alternative Feature Composition

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

6 A Note on Feature Composition

When introducing our model of features, there is one important design decision for composing features: we assume that features are composed in a particular order. 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.

Secondly, there are problems when viewing features as unordered citizens. We show in the following that, although intuitive, the idea of treating features without order is difficult wrt. 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 4. 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 original 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.

In cases where features are fully independent it is not needed to order them. But still, there is no harm with an ordering in this case, and possibly a simple syntactic extension may alleviate the problem.

6.1 Related Work

We briefly compare the feature model to other approaches. Apart from the detailed comparison, we argue that the feature model provides maximal flexibility (with static typing) and is as simple as possible.

- 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. Instead, we can just select features here.
- Method combination with before, after and around messages in CLOS [8] 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 [1] 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.
- Several other approaches allow to change class membership dynamically or propose other compositions mechanisms [9, 20, 4, 16, 10]. Note that one of the main ingredients for feature-oriented programming, lifting to a context, can also be found in [16]. 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.

7 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 not been addressed.

- An aspect not yet considered is hiding. 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. Such hidings can easily be provided by adding an appropriate interface and by disabling the others.
- Generic liftings via higher-order functions are possible in Pizza. In the stacks example it is easy to see that lifting to lock is schematic. It is natural to express this by higher-order functions. 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

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

This can be made to a default lifter, which is applied if no explicit lifters are provided.

- Another extension is to consider exception handling as a feature which can be added as needed. This is explored with (monadic) functional programming in [13, 14] and with first examples in Java in [15].

8 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, and overwriting as in inheritance is accomplished by resolving feature interactions.

The recent interest in feature interactions, mostly stemming from multimedia applications [21, 3], 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.

Compared to 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.

Acknowledgments. The author is indebted to the ECOOP reviewers for their helpful efforts to improve the paper. Also, M. Broy, B. Rumpe, and C. Klein contributed comments on earlier versions of this paper. M. Odersky made this paper possible by providing the Pizza compiler just in time.

References

1. Lodewijk Bergmans and Mehmet Akşit. Composing synchronization and real-time constraints. *Journal of Parallel and Distributed Computing*, 36(1):32–52, 10 July 1996.
2. Gilad Bracha and William Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).
3. K. E. Cheng and T. Ohta, editors. *Feature Interactions in Telecommunications III.* IOS Press, Tokyo, Japan, Oct 1995.
4. H. J. Fröhlich. Prototype of a run-time adaptable object-oriented system. In *PSI '96 (Perspectives of System Informatics)*, Akademgorodok, 1996. Springer-LNCS.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Micro-Architectures for Reusable Object-Oriented Design.* Addison Wesley, Reading, MA, 1994.
6. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, September 1996.
7. R. E. Johnson and J. M. Zweig. Delegation in C++. *J. of Object-Oriented Programming*, 4(3), November 1991.
8. Jo A. Lawless and M. Molly. *Understanding CLOS: the Common LISP object system.* Digital Press, Nashua, NH, 1991.
9. Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley, Reading, 1993.
10. Mira Mezini. Dynamic object modification without name collisions. In *this volume*, 1997.
11. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. 24th ACM Symposium on Principles of Programming Languages*, January 1997.
12. W. F. Opdyke and R. J. Johnson. Refactoring: An Aid in Designing Application Frameworks. In *Proceedings of the Symposium on Object-Oriented Programming emphasizing Practical Applications.* ACM-SIGPLAN, September 1990.
13. 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.
14. Christian Prehofer. From inheritance to feature interaction or composing monads. Technical report, TU München, 1997. to appear.
15. Christian Prehofer. An object-oriented approach to feature interaction. In *Fourth IEEE Workshop on Feature Interactions in Telecommunications networks and distributed systems*, 1997. to appear.

16. 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.
17. Lynn A. Stein. Delegation is inheritance. *ACM SIGPLAN Notices*, 22(12):138–146, December 1987.
18. Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested Mixin-Methods in Agora. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, Kaiserslautern, Germany, July 1993. Springer-Verlag.
19. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, 1991. 2nd edition.
20. David Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp and symbolic computation*, 3(3), 1991.
21. P. Zave. Feature interactions and formal specifications in telecommunications. *IEEE Computer*, XXVI(8), August 1993.