Generic Packages with Expandable Classes compared with similar approaches

Fredrik Sørensen and Stein Krogdahl
Department of Informatics, University of Oslo
E-mail: fredrso@ifi.uio.no, steinkr@ifi.uio.no

Abstract

This paper gives a short introduction to the GePEC system and to the notion of generic packages with expandable classes. It also compares this system to other languages and mechanisms with respect to some well known challenges in object-oriented programming. As the representatives for other languages we have chosen languages based on the virtual class approach. More specifically, these are J&, with its nested inheritance, and the more known CaesarJ.

1 Introduction

An important type mechanism in programming languages is one that allows the programmer to write units of code that in a flexible way can be used in different programs or parts of a program. Both procedures and classes are mechanisms with such a purpose, but similar mechanisms meant for larger structures such as frameworks, are also needed. In O-O languages such functional units will often consist of several co-operating classes and subclasses, and they will therefore often take the form of a class with inner classes or a package.

For such a package to be usable in as many settings as possible, it is important that it can be tailored to the different environments. The classes could for example need extra attributes, the typing may have to be adjusted, and one might want some renaming. Our group has worked on a mechanism called GePEC (Generic Packages with Expandable Classes) [14] meant to fill such a need. However, there are a number of other similar mechanisms, and, like the ones we will discuss, several are based on mechanisms similar to virtual classes [15, 16]. The purpose of the current work is to compare GePEC with those of these mechanisms that seem the most promising. This is, first of all, the languages J& (pronounced "Jet") [17] and CaesarJ [1].

In this paper, we first present the main features of GePEC. Then we show some examples of GePEC in action. After that we present the corresponding features in J& and compare these with those of GePEC. We also discuss some differences between J& and CaesarJ. Finally we discuss in what way generic packages differ from the other approaches.

2 An Introduction to GePEC

The GePEC system ("Generic Packages with Expandable Classes") is presented in [14], and its main mechanism, besides the traditional O-O mechanisms found e.g. in C# [12] or Java [11], is the concept of a generic package. Such a package can be "instantiated" at compile time, thus forming a new normal package as we for example know it from Java. The same generic

This paper was presented at the NIK-2007 conference; see http://www.nik.no/.

package can be instantiated multiple times in the same program, and each time produces a fully new package. Generic packages can also be instantiated inside other generic packages.

As part of an instantiation, the produced package can be tailored to its environment, so that it will fill its role there as well as possible. The most important of these tailoring mechanisms are: Extra attributes may be added to the classes, the class names may be changed, and the generic packages themselves may have type parameters that must be given actual types during instantiation. Of these, we shall mainly look at the first two in this paper. GePEC also has a general renaming mechanism, and a mechanism for regulating visibility of names, but none of these will be discussed here. The following is an example of a generic package with functionality for graphs:

```
genpack Graph {
  class Node {
    Edge firstEdge;
    Edge insertEdgeTo(Node to) { ... }
}
  class Edge {
    Node from, to;
    Edge nextEdge;
    void deleteMe( ) { ... }
} }
```

Note that we, as above, write both normal and generic packages in a form where the classes of the (generic) package appears in curly brackets. A very simple usage of this generic package is to instantiate it so that it supports a network of roads and cities needed in some program. We then want the class Node to be "expanded" to a class City with some extra attributes, and Edge to be expanded to Road. Such an instantiation can be done as follows, here inside a normal package MyProgram:

```
package MyProgram {
  CRGraph: inst Graph with Node => City, Edge => Road;
  class City adds { String name; void someMethod( ) { ... } }
  class Road adds { int length; void someOtherMethod( ) { ... } }
  class MyProgram {
    static void main(String args[]){
      ...
      City oslo = new City(); oslo.name = "Oslo";
      City bergen = new City(); bergen.name = "Bergen";
      Road osloBergen = oslo.insertEdgeTo(bergen);
      System.print(osloBergen.to.name); // Outputs "Bergen"
... } } }
```

Here the generic package Graph is instantiated (by the keyword `inst`) and a normal package, named CRGraph, is created at compile time. There are rules for the scope of names like CRGraph, but we shall not discuss these details here. The classes of the new package becomes immediately visible (as if it was imported), and as long as the names of the classes in different packages being instantiated are unique, one does not need to use the given package names at all. Their main purpose is for qualification when we have name clashes. We shall in this paper always give names (like CRGraph) to instantiations, even if this is not necessary in general.

During the above instantiation, all occurrences of the name Node in the generic package are changed to City, and likewise for Edge and Road (except for some detailed rules for constructors etc.). In addition, the classes City and Road get the additional attributes given after the keyword `adds`. Thus, the statements given in the method main will compile without

errors. In the last line e.g. “osloBergen.to.name” is OK as “to” in Edge is now retyped to City (and thus has the attribute "name").

Note that the classes originating from Graph, with the associated expansion classes City and Road, are now typed as if the whole City/Road system was written especially for the purpose of this program, and that the classes Node and Edge have totally disappeared in the resulting program. Even statements like “new Node(...)” have been changed to “new City(...)”, so that no instance of class Node (or Edge) will ever appear during program execution.

To demonstrate more of GePEC s potential, we now assume that we in addition to the generic package Graph have a generic package GeoData with classes CityData and RoadData, with similar properties as the earlier mentioned City and Road. It may look as follows:

```
genpack GeoData{
  class CityData{ String name; void someMethod( ) { ... } }
  class RoadData{ int length; void someOtherMethod( ) { ... } } }
```

We can now combine these two generic packages to form classes City and Road very similar to those obtained earlier. If Graph and GeoData had been ordinary (not generic) packages we would have needed multiple inheritance to obtain this. However, when the packages are generic, we can do this more directly by making one new class City, which is an expansion class for both Node in Graph and for CityData in GeoData. This is expressed in GePEC as follows:

```
package MyProgram {
  CRGraph:   inst Graph with Node => City, Edge => Road;
  CRGeoData: inst GeoData with CityData=> City, RoadData => Road;
  class City adds { ... more attributes? ... }
  class Road adds { ... more attributes? ... }
  class MyProgram {
    static void main(String a[]){...System.print(osloBergen.to.name);}
} }
```

The new class City will e.g. now consist of all the attributes from Node (with types Node/Edge replaced by types City/Road), and all the attributes of CityData (with types CityData/RoadData replaced by types City/Road). Thus, the statements of method `main(...)` will compile without errors. There are detailed rules for how constructors of City and Road should be written. If the same name is declared in the classes Node and CityData, they can be distinguished (from within the the `adds`-part of City) by qualifying them with the names CRGraph and CRGeoData.

Currently, GePEC does not offer general multiple inheritance (but the above mechanism can be seen as a sort of “multiple static inheritance”, where e.g. City is a very special “static subclass” of Node and CityData (“static” here meaning that it can be fully taken care of at compile time)). However, avoiding general multiple inheritance when we allow this kind of multiple static inheritance leads to some awkward rules, so this is currently under evaluation.

Also, one should note that a generic package can contain a hierarchy of classes and also that for each instantiation of a package we get a new set of all the static variables of the classes in the package.

3 GePEC in Action

We have seen how GePEC can handle some simple problems in a straight forward way. In this section, we shall look at some more involved problems. In section 4 we shall see how some similar systems based on virtual classes (J& and CeasarJ), can solve these problems.

The Expression Problem

The "expression problem" is discussed in several papers, and a good exposition can be found in [23]. The problem is connected to describing the node classes of an abstract syntax tree (AST) for a grammar, and an expression grammar is the usual example. We use the following first version of the grammar:

```
Exp   ->   Add  |  NUM_LITERAL
Add   ->   Exp '+' Exp
```

For the AST of this grammar we can write the node classes below. The problem itself occurs when we want to extend this AST in different ways, either by adding a new node (e.g. for negation), or by adding a new procedure (e.g. print) that must have specific versions in each node class. In [23] it is explained why it is not easy in standard o-o languages to allow both of these extensions in an unintrusive way, such that we end up with satisfactory typing. If we simply implement the methods (like eval and print) in the node classes themselves it is easy to add a new node class, while if we use the visitor-pattern [10] it is easy to add a new method. Below is the AST package.

```
genpack Expression {
  class Exp { int print(); ... }  // We skip abstract etc.
  class Add extends Exp {
       Exp left, right;
       int eval(){ return left.eval() + right.eval(); } }
  class Literal extends Exp { int value; ... } }
```

If we now extend the grammar with the two productions:

```
   Exp  ->   Neg
   Neg  ->   '-' Exp
```

we can obtain a corresponding extension of the AST nodes with the following new generic package, instantiating the old one:

```
genpack NegExpression{
  NE: inst Expression with Exp => NegExp,
                           Add => NegAdd,
                           Literal => NegLiteral;
  class NegNeg extends Exp { int eval(){ return - expr.eval();} }
```

Note that we could have reused the name Exp (instead of introducing NegExp etc.), but for clarity we have used new names. Now, instantiating NegExpression in our program instead of Expression will give us what we want.

In fact, for doing just the above, we did not need generic packages at all. But if we want to add e.g. a new method print, to have the original version of Expression as a generic package is essential:

```
genpack PrintExpression{
  PE: inst Expression with Exp => PrintExp,
                           Add => PrintAdd,
                           Literal => PrintLiteral;
  class PrintExp adds { void print(); }
  class PrintAdd adds { void print(){ left.print(); output("+");
                                      right.print(); } }
  class PrintLiteral adds {void print(){ output( value ); } }
}
```

Note that if we did not use GePEC, but made PrintExp a normal subclass of Exp PrintLiteral would not be a subclass of PrintExp. By using the expansion mechanism, we only add new attributes to the old classes (and rename them) so that the old hierarchy is retained. Equally important, in the old code declarations typed with e.g. Exp would not have changed type to PrintExp.

Finally, an observation concerning GePEC s current limitations: One could hope that if NegExpression and PrintExpression somehow were instantiated “together”, we could obtain one hierarchy of AST classes, that had both expansions, so that we only had to add how print should work for the Neg class. This is currently not possible, and we therefore would have to do the expansions one after the other, e.g. by letting PrintExpression instantiate NegExpression. However, as we shall see in section 4, other languages have this ability.

Lists and Lists of Lists

A generic package may be instantiated several times in the same program, and the produced packages can be tailored quite differently and will be totally independent of each other. We assume the following package for forming lists:

```
genpack Lists {
  class List { Element first, last; ... }
  class Element { Element next; ... } }
```

A program that will use this genpack to implement lists of persons and lists of cars could look like this:

```
package Program {
  PL: inst Lists with List => PersonList, Element => Person;
  CL: inst Lists with List => CarList, Element => Car;
  Class Person adds { String name; int age; ... }
  // And likewise for PersonList, CarList and Car
  // We now have two separate list systems, with classes (PersonList,
  // Person) and classes (CarList,Car).
  // These are kept fully separate by the compiler.
}
```

For a more complicated example we can demonstrate how the generic package Lists can be instantiated twice to produce lists of lists, here used to implement a sparse matrix (also implemented as a generic package to make further tailoring possible, e.g. by adding a value in the class Item). Below, the classes from the imported packages are combined so that what is an element in one instance (Column in LSM) is a list in the other (Column in LCI).

```
genpack Matrix {
  LSM: inst Lists with List => SparseMatrix, Element => Column;
  LCI: inst Lists with List => Column, Element => Item;
  class SparseMatrix adds { ...
    Item getItem(int column, int row){
      Column col = first;
      while ( col != null && col.cNo != column){ col = col.next; }
      ... and so on ...
    }
   class Column adds { int cNo; }
   class Item adds { int rNo; } }
```

4 The J& Language

J& [17] is another language with mechanisms for writing sets of classes so that the whole set can be tailored to the actual environment when it is used. In J&, such a set of classes can be held together either by an "outer class" (so that the inner classes makes up the set), or in a package. The tailoring is done by defining a subclass of the outer class or by defining a "subpackage". Here, classes with the same name as those in the superclass/superpackage will be considered as additions to these classes. Mechanisms of this type are usually referred to as "virtual classes", and they were pioneered by Beta [16, 15].

For our examples below, we shall use an outer class (and not a package) to frame the set of classes. In J&, we can write a Graph "package" corresponding to the one we wrote in GePEC as follows:

```
class Graph {
   class Node{ Edge firstEdge; Edge insertEdgeTo(Node to) {...}}
   class Edge{ Node from, to; Edge nextEdge; void deleteMe(){...}} }
```

In J& we can also create a similar program as we wrote in GePEC with additions for City and Road. It must be written in a subclass of Graph, and could look as follows:

```
class MyProgram extends Graph {
   class Node { String name; void someMethod( ) { ... } }
   class Edge { int length; void someOtherMethod( ) { ... } }
   class Main{
     void main(String args[]){
       ...
       Node oslo = new Node(); oslo.name = "Oslo";
       Node bergen = new Node(); bergen.name = "Bergen";
       Edge osloBergen = oslo.insertEdgeTo(bergen);
       System.print(osloBergen.to.name); // Outputs "Bergen"
... } } }
```

As name correspondence is used to decide which class in Myprogram is considered an addition to a class in Graph, we have to stick to the class names Node and Edge (instead of introducing City and Road). In many situations, that is obviously a drawback.

J& allows general multiple inheritance where a common superclass (along different paths in the inheritance graph) leads to shared variables. Thus, if we, as in the earlier GePEC example, have a J& class GeoData, with inner classes CityData and RoadData, we cannot combine it with the Node/City and Edge/Road, as the names are not the same. We could, however, hope that it would work if we changed the name of the classes CityData and RoadData to Node and Edge, and used the following construct:

```
class myprogram extends Graph & GeoData {
  class Node {... something more? ...}
  class Edge {... something more? ...} }
```

However, this will not combine in the way we wanted, as there are special requirements for this to happen. Slightly simplified these say that such a combination only happens for those names (like e.g. Node) that stems from a common version defined in some common superclass of Graph and GeoData. Thus, this sort of combination has to be planned in advance, both concerning the choice of names, and concerning superclasses. In GePEC one is much freer in both these respects.

The Expression Problem and J&

We shall look at the expression example again, and it turns out that in this case the requirements in J& for combining classes are naturally satisfied so that we to a large extent can get the same effect as with generic packages, and in one sense even a little more. We can write the following base class (or package) for expressions.

```
class Expression {
  class Exp { ... }
  class Add extends Exp { int evaluate(){ ... } }
  class Literal extends Exp { int evaluate(){ ... } }
  class Compiler {
    void main() { ... }    Exp parse(){ ... } } }
```

To extend it with a class negation we write:

```
class NegExpression extends Expression {
  class Neg extends Exp { int evaluate(){ ... } }
  class Compiler { ... } }
```

We also have an extension for printing

```
class PrintExpression extends Expression {
  class Exp { void print(){ ... } }
  class Add extends Exp { void print(){ ... } }
  class Literal extends Exp { void print(){ ... } }
  class Compiler { ... } }
```

We might want to combine these two, which can be done as follows:

```
class NegPrint extends NegExpression & PrintExpression {
  class Neg { void print(){ ... } }
  class Compiler {
    void main() { // New main.
      Exp e = parse();
      int val = e.evaluate();
      e.print();
    }
    void parse(){ ... } // New parse.
} }
```

In J&, typenames are usually only a shorthand notation for what is called "dependent classes" and "prefix types", so the real code is more complex than it is written. A prefix type, written `P[T]`, is the enclosing class of the class `T` that is a subtype of `P`. The name `Exp` in base is therefore an abbreviation for `base[this.class].Exp` (where `this` has the usual meaning), which for a runtime instance of for example `NegPrint.Neg` would be `NegPrint.Exp` (Note that this particular class is not explicitly defined in the class `NegPrint`). This gives J& a late binding of types.

So, unlike with generic packages, where typenames are resolved statically for instantiations of generic packages, types in J& are really defined as being relative to the runtime instance of an object. These dependent types are not needed with generic packages. Generic packages may thus be seen as providing a type system that is easier to work with.

In the example, we see that late binding in J& also applies to the superclass declarations. This is what in J& is called a virtual superclass and it is what allows one to reuse the entire inheritance tree of one package in another. What is meant by this is that `NegPrint.Neg` is a subclass of `NegPrint.Exp`. This is in some ways similar to what happens with instances of generic packages and the new types created there.

As mentioned earlier, when combining the two extensions we cannot get this kind of solution with generic packages, because GePEC insists on keeping the instances of the common base package completely separate from each other, also if the two extensions are combined. Had we written the example above (syntax adapted) in GePEC, the result would have been that in `NegPrint` we would have two different classes Exp and two different classes Add and so on from the two different packages NegExpression and PrintExpression. Also, combining them into one like List and Element (as Column) is in the lists of lists example) would not yield the right result, since there would be separate instances of the procedures in them. So, for the two different `Literal`-s, there would be two `value` attributes in the combined class just like there would be both next and head/tail variables in the Element class in the lists of lists example.

When one wants to adapt a package to a given environment, generic packages allows one to be very free, since everything will be typed with types in the new context and names can be changed to accommodate the situation. The instance of the package will also be completely independent of the other instances of the same generic package, and therefore one can locally think of having everything to oneself. Also, one does not have to worry about objects of the classes of the base (generic) packages with generic packages, since there will only be objects of the final classes (classes of non-generic packages), and never objects of the classes in the generic packages. Objects from different instantiations of a generic package will never get mixed up since they are considered to be of completely different types.

There is an ongoing discussion within our group: Should GePEC have a mechanism like the package extension mechanisms in J& or something else that resembles "virtual superclasses" in C++.

The Lists of Lists Example and J&

We would also like to be able to write something similar to the lists of lists example in J&, which would be to use the same generic package more than once in the same scope and get two different "instances" of it with different types. We would like to avoid casting or qualified type names.

It is straight forward to write an outer class List in J& that has as its inner classes the same classes as the package List in the GePEC example. To show how the mechanisms are different we could simply try to use it twice in J& as we did in GePEC.

We cannot combine types and change typenames in the same way with J& as we can with expandable classes. We have to choose between using the same name (which is not an option with lists of lists) or use (regular) multiple inheritance.

An Element is still en Element in such a package and we have the same names as in the Lists package, but we have introduced subclasses. We would then have to use type casting to get a Column or an Item variable to point to an element from the lists (from i.e. the `to`-pointers) and since a variabel typed with Element can point to both a Column and an Item it does not give us the type safety that we would get with generic packages.

5 CeasarJ

CeasarJ [1] also has similarities with generic packages and the way one extends and combines modules in CaesarJ is very similar to J&.

In CaesarJ there are not packages but only classes with inner classes. These CaesarJ classes are called *cclasses*. Like J&, CaesarJ also has an operation for merging classes. The difference is that one may only merge *cclasses* and not classes in general. Since they may have inner

classes, at the outmost level it is similar to merging packages in J&. CaesarJ also has an `&` operator, and it works in much the same way as in J&.

When a *cclass* is created by joining two or more CaesarJ classes there is no way to choose which classes are combined together. Like for J&, it is done by combining those with a common superclass only. When using this form of multiple inheritance in CaesarJ, the inheritance graph (which may include common ancestors) is linearized. Therefore, it may, like mixins [6], destroy a useful parent-child relationship in terms of the use of super. The linearization does not allow one to write something like the example above with the lists of lists. The linearization also seems to make it more difficult to reason about a package (cclass) written for reuse and also to read and understand a package.

In [1], they also focus on how useful it is to combine different extensions of the same base package, and do not discuss extensions in general. Most of the arguments about J& compared with GePEC also applies to CaesarJ. In CaesarJ, like in J&, one cannot as easily adapt a package to a given situation by changing names and so on. Neither can one choose which classes to combine. One can also not keep different instances of a package from each other. Aspects are supported in CaesarJ, but not in J& and GePEC.

6 Discussion

In this study we have looked at the following problems, some of which have also been brought up by others.

- Combining completely different packages (*Geography problem*). Problems like this and its solutions where brought up by Krogdahl [14] and others [18, 22] The problem is to combine packages (or classes with inner classes) that have been written separately, and to be able to join together classes from them and get the combined functionality of the classes. It should be done with as little "glue" code as possible, only describing what was not described in either or resolve ambiguities.
- Using the same base package in different ways in the same scope. (e.g. *Lists of lists problem*) They may involve different types in the new scope for the same "concept" in the base package. This is what is done in the list of lists, where the element in one list is the head of the other. One may note that simpler versions of these problems are often solved with generic classes or functions (parameterized types) [7, 2]
- "Orthogonal" extensions (*Expression problem*). This occurs when one wants to extend something in different "directions" when the base was not written with this in mind. How can we prepare something for such extensions? The expression problem is the "classic" example, where one may want to add both new nodes (neg, solved with regular inheritance) and new "passes" (print, solved with the Visitor pattern). When both kinds of extensions are wanted something else is needed.
- Composing extensions as in the J& compiler example. This is also discussed in [17] (expressions) and [1] (displays). The problem is that a base system has been extended in different ways, which may include orthogonal extensions like with the expressions, and one wants to "join" them so that the common base is a shared common version and all extensions are added in some way. Of course, ambiguities must be resolved.

We have seen in the examples that different languages can solve the above problems to a different extent. It turned out that the "virtual class approach" of J& with the join operator `&` is rather restricted compared with generic packages. On the other hand, generic packages

has limitations related to combining different extensions to the same base. The differences are mainly related to the power of expression one has for these purposes. Below we discuss some points related to this.

- *How easily can one adjust a module to a given situation?*. Neither J& nor CaesarJ have the same ability as generic packages to adjust the package to the given situation. They do not have the same mechanisms for renaming and for combining completely separate classes from different sources in the same way as one has with generic packages. We have seen in the examples (and others have described) how such expressiveness is useful.
- *How simple and intuitive are they to use?* (How complex / simple is the type system.) Mechanisms like the ones in J& and CaesarJ have very complex type systems and they may be less easy to understand than a static instantiation of a package. Because of the dependent types of J& and the linearization of inheritance in CaesarJ, it may also be less easy than with generic packages to reason about the (generic) packages before they are used. Also, one knows that there may be some conflict between the objects of the base (generic) packages and the packages that use them (clients) which for the "virtual class" mechanisms are resolved e.g. with the use of `final`.
- *Does it allow many fully independent usages of the module in some program?* With generic packages, as we have seen with lists of lists, completely different instances are created of a generic package for each new use and new types are created each time. This has a lot of power that one cannot get from mechanisms where there is a "shared" common base.
- *Joining separate extensions.* When one wants the kind of solution that we saw when we combined the `NegExpression` and `PrintExpression` packages in J&, both J& and CaesarJ solves this problem really nicely. The way their **&** operators work is fine for examples like that. But, there are many other ways that we may want to combine packages and for many of those, we want the generic packages kind.
- *How a change to the implementation of a package affects "clients".* Also, there may be a provider of a package (P) that uses a generic package (GP1) for its implementation and a client that uses the package (P) in its program (C). The client will not be affected by a switch by the provider to an implementation that uses a different generic package (GP2) as long as the package does not change in other ways (type and method names, and of course semantics). Also, the client may import and use the generic packages (GP1, GP2) in their program (C) without interference between the "instances" of the packages. This may not always be the case with the other approaches.

7 Related Work

A lot of work has been done in the area of finding new solutions to writing reusable pieces of code (frameworks) and combining them and also combining extensions of the same base. All of these try to add something to the power of inheritance.

A very important contribution was virtual classes in *Beta* [16] [15]. *Feature-Oriented programming* [19] focuses on writing the difference between a pair of features instead of writing the code that binds together a set of different classes. This reduces the amount of glue code to be written for a set of features that one wants to use in different combinations, since only code for each pair of features is needed and not for all the possible combinations. *Mixins* [6] are a way of adding several superclasses to a class in such a way that they become sub- and super-classes of each other. *Traits* [9, 20] are a way of joining together different pieces of a

class (the traits) when creating a class in such a way that they (the traits) do not have sub- and super-class relationships. State is only provided in the class and never in the traits. *Multiple inheritance* [13, 21] allows one to combine classes from different sources. *Open Classes* as in MultiJava [8] allows one to add a set of methods to a class without creating a subclass. Combining inheritance hierarchies have been discussed in [18] and with *hyperslices* in [22]. *Classboxes* [3, 4] is a language that provides method addition and replacement, while keeping the changes made local to the classbox. Some of the problems can of course also be solved with *generics* [5].

8 Conclusion and Further work

When writing and using some kind of framework there has been acknowledged that there is a need for something more than regular inheritance and a lot of mechanisms and languages have been proposed. The basis for many of these mechanisms are virtual classes. What we have found is that there is a significant difference in what kind of expressiveness one gets from these different languages and that some of the expressiveness that one gets with generic packages, and not that easily with virtual classes, may be very useful in writing and using frameworks.

This work is part of the SWAT project (The Research Council of Norway grant no. 167172/V30).

References

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer, 2006.

[2] Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for java. In *POPL 97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 132–145, New York, NY, USA, 1997. ACM Press.

[3] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/j: controlling the scope of change in java. *SIGPLAN Not.*, 40(10):177–189, 2005.

[4] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of Joint Modular Languages Conference (JMLC 03)*, volume 2789 of *LNCS*, pages 122–131. Springer-Verlag, 2003.

[5] Gilad Bracha. Generics in the java programming language. Technical report, Sun Microsystems, Inc., July 2004. Url: `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`.

[6] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP 90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[7] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA 98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[8] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA 00: Proceedings of 15th ACM SIGPLAN conference on OO-programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[9] Gael Curry, Larry Baer, Daniel Lipkie, and Bruce Lee. Traits: An approach to multiple-inheritance subclassing. In *Proceedings of the SIGOA conference on Office information systems*, pages 1–9, 1982.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[12] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Professional, 2nd edition, 2006.

[13] Stein Krogdahl. Multiple inheritance in simula-like languages. *BIT*, 25(2):318–326, 1985.

[14] Stein Krogdahl. Generic packages and expandable classes. Research Report 298, Department of Informatics, University of Oslo, October 2001.

[15] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA 89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM Press.

[16] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[17] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In Peri L. Tarr and William R. Cook, editors, *OOPSLA - Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, pages 21–36. ACM, 2006.

[18] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *OOPSLA 92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 25–40, New York, NY, USA, 1992. ACM Press.

[19] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP 97 - Object-Oriented Programming*, volume 1241. Springer, 1997.

[20] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. In *ECOOP*, pages 248–274, 2003.

[21] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, Helsinki, 1987.

[22] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE 99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[23] Mads Torgersen. The expression problem revisited. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, volume 3086 of *Lecture Notes in Computer Science*, pages 123–143. Springer, 2004.