Adaptive Code Reuse by Aspects, Cloning and Renaming

Anya Helene Bagge
Martin Bravenboer
Karl Trygve Kalleberg
Koen Muilwijk
Eelco Visser

Technical Report UU-CS-2005-031

Department of Information and Computing Sciences
Universiteit Utrecht

August, 2005

Address:

Department of Information and Computing Sciences
Universiteit Utrecht
P.O.Box 80089
3508 TB Utrecht
The Netherlands

Martin Bravenboer
`martin@cs.uu.nl`
`http://www.cs.uu.nl/~martin`

Koen Muilwijk
`pmuilwi@cs.uu.nl`
`http://www.cs.uu.nl/wiki/Students/KoenMuilwijk`

Eelco Visser
`visser@acm.org`
`http://www.cs.uu.nl/~visser`
(corresponding author)

Department of Informatics
University of Bergen
Postboks 7800
N-5020 Bergen
Norway

Anya Helene Bagge
`anya@ii.uib.no`
`http://www.ii.uib.no/~anya`

Karl Trygve Kalleberg
`karltk@ii.uib.no`
`http://www.ii.uib.no/~karltk`

Adaptive Code Reuse by Aspects, Cloning and Renaming

Anya Helene Bagge[2,1] anya@ii.uib.no
Martin Bravenboer[1] martin@cs.uu.nl
Karl Trygve Kalleberg[2,1] karltk@ii.uib.no

Koen Muilwijk[1] pmuilwij@cs.uu.nl
Eelco Visser[1] visser@acm.org

(1) Institute of Information and Computing Sciences
Universiteit Utrecht, P.O. Box 80089
3508 TB Utrecht, The Netherlands

(2) Department of Informatics
University of Bergen, PB 7800
N-5020 Bergen, Norway

Abstract

Effective code reuse is desirable, but difficult to achieve in practice, since it is often necessary to adapt code before it can be reused successfully. The good old solution to code reuse is simple: copy, paste, then edit as needed. This is a brilliant idea, except for the maintenance problems it causes. In this paper we introduce a language extension for declaratively performing adaptive code reuse at compile-time. We decompose reuse into two operations; *clone* existing code, and *adapt* it to new requirements. The clone and adapt technique allows flexible code reuse, untangled from subtyping and other irrelevant features, and without the maintenance nightmare of copy&paste programming.

1 Introduction

Writing reusable code is notoriously difficult and code reuse through inheritance is not sufficient to solve this. A fundamental requirement for reuse is the ability to perform code composition from reusable parts. The programmer of these parts must try to anticipate future reuse scenarios by designing in variation points which make them easily composable. Even when an effort has been made to make the code reusable, further adjustment may be necessary. Adaptation may be done by editing the code, by using inheritance, aspects or some other program transformation technique.

Inheritance is often the preferred way of doing code reuse in object-oriented systems. The details depend on the programming language, but inheritance typically allows you to add new fields and methods to a class, and also override existing methods. However, the inheritance concept is actually meant for modelling an *is-a* relationship between classes; it implies a subtype relationship between the original class and the new class (allowing instances of the new class to be used instead of instances of the original class), which is not always desired. Mixins [8] have been suggested as an alternative formulation of inheritance which is believed to be more flexible, as it offers increased composability compared to single inheritance.

While mixins and inheritance provide mechanisms for code *composition*, aspects [20] provide a good way for doing code *adaptation* by selectively weaving pieces of extended behaviour into existing classes. The programmer has detailed control of this weaving through a high-level, declarative language. However, the in-place modification to existing code restricts its use as a general code adaptation mechanism, as we will show.

Another very useful adaptation technique is intertype declarations. Like open classes [10], this language feature of AspectJ allows the introduction of new interfaces, methods and fields into existing classes and is in this sense competing with inheritance and mixins. As with aspects, it is also hampered by restrictions related to in-place modification.

It would appear that mixins and aspects should together offer an appealing solution to both composition and adaptation of code by allowing the reusable parts to be expressed and composed using mixins, with further adaptation performed by aspects.

We observe that by extending intertype declarations with the ability to clone classes, essentially lifting the in-place restriction, we have exactly what is needed to implement mixins. The question then becomes: if we add this cloning as a separate language construct, can it combine favourably with more than just intertype declarations?

In this paper, we describe a declarative extension to the Java language (but it is generally applicable to any object-oriented language) that provides cloning with renaming as a language construct. By combining cloning with aspects (for code adaptation), we obtain a wide range of reuse scenarios.

We will show that cloning offers clear and concise solutions to design problems which cannot be expressed cleanly by plain object-orientation or aspects. Furthermore, we will show that the clone operator is a fundamental ingredient in many popular language features such as templates and mixins and that it also augments aspects by making them more suitable for expressing generative programming.

The article is organised as follows. In Section 2, we explain the clone operator. In Section 3, we show how the clone operator may be combined with renaming, aspects and intertype declarations to allow per-context weaving, parameterisation of classes and reusable implementations of design patterns. In Section 4, we show how the clone operator combines with intertype declarations to obtain mixins. In Section 5 we relate our work to other language features and techniques for reuse in object-oriented systems. In Section 6 we discuss the implications of the clone operator, and in Section 7 we conclude and point out further work.

2 Clone and Rename

In this section we describe a declarative extension to the Java language which allows decoupling of code reuse from subtyping. However, the concept is not restricted to Java and is generally applicable to other object-oriented languages. The extension provides a *clone* operator that can be used to make a copy of a particular set of named definitions: packages, classes, fields and methods.[1] The clones receive their own names.

```
Clone      ::= Visibility? clone Definition
               Direction? as RenameExpr
               WithClause*
Direction  ::= + | -
WithClause ::= with Aspect
             | with Identifier = Identifier
             | with MethodSig as MethodSig
Definition ::= PackageName
             | ClassName
             | MethodName
RenameExpr ::= Identifier
             | Identifier *
             | * Identifier
             | Identifier * Identifier
Aspect     ::= aspect AspectBody
             | aspect AspectName
MethodSig  ::= Visibility Type MethodName
               (Type , ...)
Visibility ::= private | protected | package
             | public
```

Figure 2: Syntax for the clone operator. *AspectName*, *PackageName*, *ClassName* and *MethodName* are all variants of *Identifier*, which have been disambiguated by the Java/AspectJ type system. *Type* is a Java type expression. *AspectBody* is the body clause of an aspect definition. In the case of *MethodSig*, one or more of *Visibility*, *Type*, and *MethodName* may wildcarded using `*`.

Consider Figure 1. In this example, `A`, `B` and `C` are classes. The class `C` inherits from `B`, which in turn inherits from `A`. The operator `clone B as B1` will make an identical copy of `B`, named `B1`. The resulting inheritance hierarchy becomes `B1 extends A`, as expected. There is no typing relation between `B1` (the clone) and `B` (its original).

For convenience, the `clone` operator is aware of inheritance. By writing `clone B+ as *2`, the user clones the class `B` and all its subclasses. Their names will all be suffixed by `2`. Similarly, a `clone B- as *3` will clone `B`, including all its parents. This makes it possible to easily clone (parts of) existing class hierarchies.

We stress that it is the class definition which is being cloned, not objects of the class.

[1]In this article, we will focus mostly on cloning at the class level.

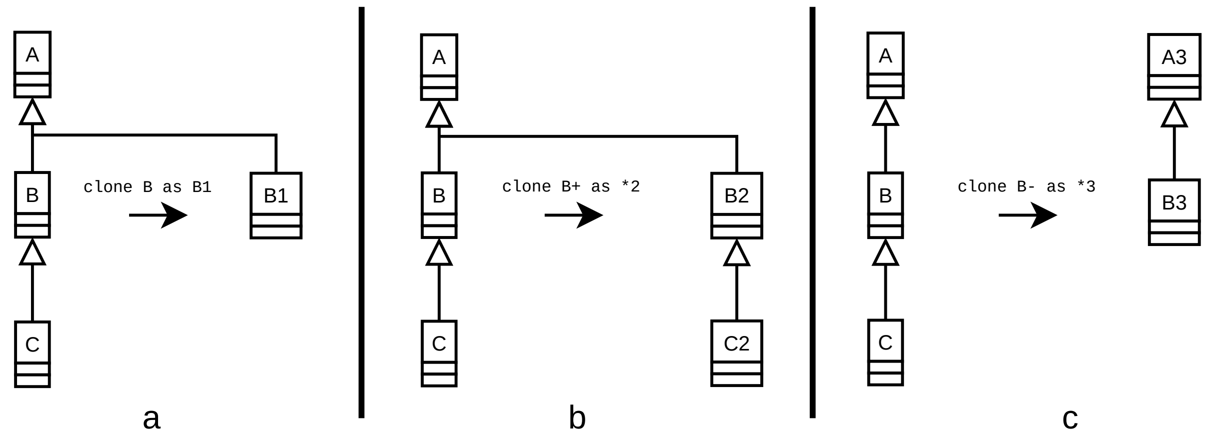

Figure 1: The clone operator applied to classes. (a) shows cloning of class `B`, (b) shows cloning of class `B` and all its subclasses, (c) shows cloning of class `B` and all its superclasses.

The syntax for the clone operator is given in Figure 2. The direction of cloning (`+` and `-`) is only defined when the selected definition is a class. The cloning operator as applied to a class works in three steps:

1. *copy*; the selection is evaluated. This will result in a set of one or more classes. E.g., in the case of (a) in Figure 1, only one class is selected, in (b) and (c) the result is multiple classes.
2. *rename*; the supplied rename expression is applied to the set obtained in (1). Details on the rename expression are given further down. For each name which is modified, a mapping from its old name to the new name is recorded.
3. *fixup of internal references*; the bodies of all methods in the selected classes are traversed, and internal references are updated, according to the map constructed in the previous step.

Essentially, steps two and three are together a rename refactoring, applied to the clone. Conceptually, the clone operator works on the name space: it can only be applied to named entities. It must always respect the name space constraint that two entities may not share a name. I.e., the resulting name of a class after cloning must not be in conflict with another class in the same package. This constraint must and can be checked at the time of cloning.

Another consideration is references to and from clones. The Java type system (and practically any static type system) does not accept undefined references in a program. The clone operator will never introduce undefined references, as internal references are resolved.

A concrete, though slightly contrived example of cloning is given in Figure 3. The example demonstrates that cloning is equivalent to copying the source code for `Component` and `Message`, then applying appropriate rename refactorings to both.

The extension requires no runtime support. It may be implemented as an independent stage in the compiler, before aspects and type checking. Alternatively, it may be implemented as a separate preprocessor. Even though cloning is primarily useful for whitebox code reuse, our implementation does not require access to the source code. Cloning with renaming of packages, classes, methods and fields works equally well at the bytecode level.

2.1 Name rewriting

Consider Figure 1 again. As part of a clone operation, the name of the class being cloned must be modified. When cloning a set of classes (using `-` or `+`), our extension allows the name to be prefixed, postfixed or both. Where only one class is cloned, its name may be replaced arbitrarily.

The name rewriting only allows changes to the local part of the name of a definition. It is not pos-

```
private clone Component+ as Cloned*;
abstract class Component {
  public final int MODAL = 1;
  private int state = 0;

  public abstract int getWidth();
  public abstract int getHeight();
  protected void setState(int s)
  { state = s; }
  public boolean isModal()
  { return state & MODAL; }
  public void paint() { ... };
}
class Message extends Component {
  private String m;
  public Message(String m)
  { this.m = m; setState(Component.MODAL); }
  public Message()
  public void setMessage(String m) { ... }
  public int getHeight() { ... }
  public int getWidth()  { ... }
  publiv void paint() { ... }
}
```

$$\Downarrow$$

```
class ClonedComponent { ... }
class ClonedMessage extends ClonedComponent {
  ...
  public Message(String m)
  { this.m = m;
     setState(ClonedComponent.MODAL); }
  ...
}
```

Figure 3: Application of `clone+` to a class hierarchy.

sible to change the package of a clone using name rewriting, nor the class of a method or field.

2.2 Changing visibility

The visibility of the clone may be changed arbitrarily from its original using the `private`, `public`, `protected`, and `package` modifiers in front of `clone`. The `package` keyword for visibility was introduced for this purpose, as Java does not have an explicit keyword for package visibility; it is assumed where no other visibility keyword is given.

2.3 Cloning and Packages

The clone extension does not allow cloning classes into new packages, or methods into new classes. The scope of the clone must match the original. In the case of classes, cloning into new packages would most likely require violation of encapsulation if the clone was to operate properly. For example, suppose the `public` class `frontend.Parser` relies on `TokenFactory`, which has package visibility. Cloning `frontend.Parser` into package `doctool.Parser` will leave undefined the reference to `TokenFactory`. It cannot be rewritten to `frontend.TokenFactory`, as that would violate encapsulation of the `frontend` package. The problem is even more acute when cloning a method from one class to another: all implicit dependencies must either also be cloned, or they must be anticipated by the developer performing cloning.

3 White Box Reuse

In this section we explore the application of the clone operator to white box code reuse. In particular, we show how cloning enhances the code adaptation that can be achieved with aspects.

3.1 Per-Context Weaving

The aspects provided by the AspectJ language [3] are applied on a per-class basis, thus affecting all objects of that class, system-wide. It is technically possible, though awkward to have multiple versions of a class in your system, each in its own context (classloader) with its own set of weaved aspects. As we shall demonstrate in the sequel, this limitation is

```
clone Message as LoggedMessage with aspect {
   pointcut messageCreation(String m) :
    initialization(LoggedMessage.new(String))
    && args(m);

  before(String m) : messageCreation {
    Logger.log(m);
  }
}
```

Figure 4: Code reuse without inheritance. Behavioural extension using aspects.

```
class Array {
  private Element data[];

  public Element getElement(int index)
  { ... }
  public void setElement(Element elt,
                         int index)
  { ... }
}
```

$\Downarrow$

```
class StringArray {
  private String data[];

  public String getElement(int index)
  { ... }
  public void setElement(String elt,
                         int index)
  { ... }
}
```

Figure 5: Instantiation of `StringArray` from `Array` by renaming `Element` to `String`.

even more problematic for intertype introductions than it is for aspects.

Run-time weaving, such as provided by CaesarJ [26], provides a solution to this problem. Aspects can be applied on a per-object instead of per-class basis, but this requires (part of) the aspect weaver to be available at runtime as well.

The code in Fig. 4 shows how per-context weaving can be achieved at compile-time using cloning. By `clone Message as LoggedMessage` we can now create `Message`s with and without logging depending on the context. Furthermore, the type system will ensure that we will not use one class instead of the other. In other words, this is useful in those cases were reuse with adaptation is required, but subtyping is not.

Other applications of per-context weaving include mixins as discussed in Section 4 and adaptation of reusable design pattern code as we will discuss later.

3.2 Implicit Parameterisation of Classes

Traditionally, Java implementations of container data types, such as lists or arrays, use `Object` as the element type. This causes problems for static type checking; there is no way for the compiler to tell the difference between a list of integers and a list of strings, and no way to ensure that all elements in a container are of the same type. Furthermore, the programmer has to use type casts when retrieving objects from the container, leading to syntactic clutter. In some languages, such as C++, this can be solved using generic programming, by parameterising the container class with the element type. You can then create a 'list of integers' and a 'list of strings', and the compiler can tell the difference.

Generic programming is supported in Java 1.5 [17, 7], but not in earlier versions of the language. Using clone and rename, we can implement a simple form of generics, akin to templates in C++. Consider the class `Array` from Fig. 5, which contains elements of type `Element`. We can make an array of strings by cloning `Array` as `StringArray`, substituting `String` for `Element`:

```
clone Array as StringArray
      with Element = String;
```

This will create a new class `StringArray` with exactly the same functionality as `Array`, except that it only allows strings as elements.

Since parameterisation is done by renaming, we are not restricted to the set of parameters the class implementor chooses to make available; any name is a potential parameter. For instance, suppose we have an old, non-generic implementation of `Array`, which uses `Object` elements. We can then instantiate type-safe versions using clone and rename:

```
clone Array as StringArray
      with Object = String;
```

New code can use the safer `StringArray`, while legacy code is unaffected, and can continue using `Array`. Furthermore, if generics is available, we can use clone/rename to turn non-generic code into generic code:

```
clone Array as GenericArray<T>
      with Object = T ;
```

This technique cannot always be applied safely. Consider a hash table class, which uses `Object` for both its elements and its keys. If we try replacing `Object` with `String` to create a hash table of string elements, the key type is also changed, which may not be appropriate. This is a weakness of renaming as an adaptation mechanism.

3.3 Design Patterns

Design patterns are recipes for solving recurring programming problems that need to be adapted to the application at hand. By definition, a pattern is a pattern (in some language) if it is not available as a language feature and cannot be abstracted into a library [1, 16]. This entails that patterns are not amenable to code reuse and require instantiating the pattern manually for each occurrence in the code. Besides the code duplication that this causes, it also means that the design decision to use the pattern is now encoded at a low-level in the program. Given the right language features, patterns can be expressed directly as reusable programs. For example, Hannemann and Kiczales [18] show that aspects can be used to render a number of the GoF design patterns [15] as reusable code. However, the lack of a cloning operator forces designs that do not correspond to the standard OO designs that would have been used without considering reuse.

In the rest of this section we discuss several examples of reusable implementations of design patterns using a combination of cloning and aspects.

3.4 Filter Collections

Filter collections are a variation on the Composite pattern. The usage scenario of filter collections is as follows. An application has objects that need processing in several ways. For each kind of processing there may be several 'filters' available. The following interfaces provide a concrete example of two types of filters. An `IProcessor` processes a message response, while an `IMessageFilter` can apply a filter to a `Message` or validate it.

```
public interface IProcessor {
  void process(Response req);
}

public interface IMessageFilter {
  void apply(Message msg);

  boolean validate(Message msg)
    throws Exception;
}
```

Example implementations of these interfaces are `TimeStampProcessor`, which adds a time-stamp to a response message, `EncryptFilter`, which encrypts the contents of a message, and `NormalizeFilter`, which normalises the headers. Note that the types of filters and the signature of the filter methods is entirely application specific. Also there is no direct relation between `IProcessor`s and `IMessageFilter`s, except that they play a similar role in the design pattern.

Now an application needs to apply several filters consecutively, or choose from a pool of available filters, where the set of filters to apply can alter dynamically. Thus, the application needs to maintain a collection of filters and several strategies for applying the filters in a collection. This is in fact a pattern that could be reused in many applications. However, in practice it needs to be recoded for every application *and* for every type of filter in the application.

With cloning and renaming we *can* factor out the reusable pattern from a `ConcreteFilterCollection` hierarchy and turn it into a reusable `FilterCollection` hierarchy. Figure 6 shows the structure of the pattern. The classes on the left are the reusable design pattern classes, the classes on the right are their concrete instantiations obtained by cloning.

The `FilterCollection` class provides functionality for maintaining a collection and an abstract `apply` method. The `apply` method is implemented in subclasses to provide alternative *strategies* for applying the filters in a collection.

Assume we clone `FilterCollection` from Figure 6 twice, once to `ChoiceCollection` and once to `SequentialCollection`. The task of the `SequentialCollection` is to apply all filters in its collection. This is handled by its `apply` method:

```
public void apply() {
  for(Iterator i = _filters.iterator();
      i.hasNext();) {
    try {
```

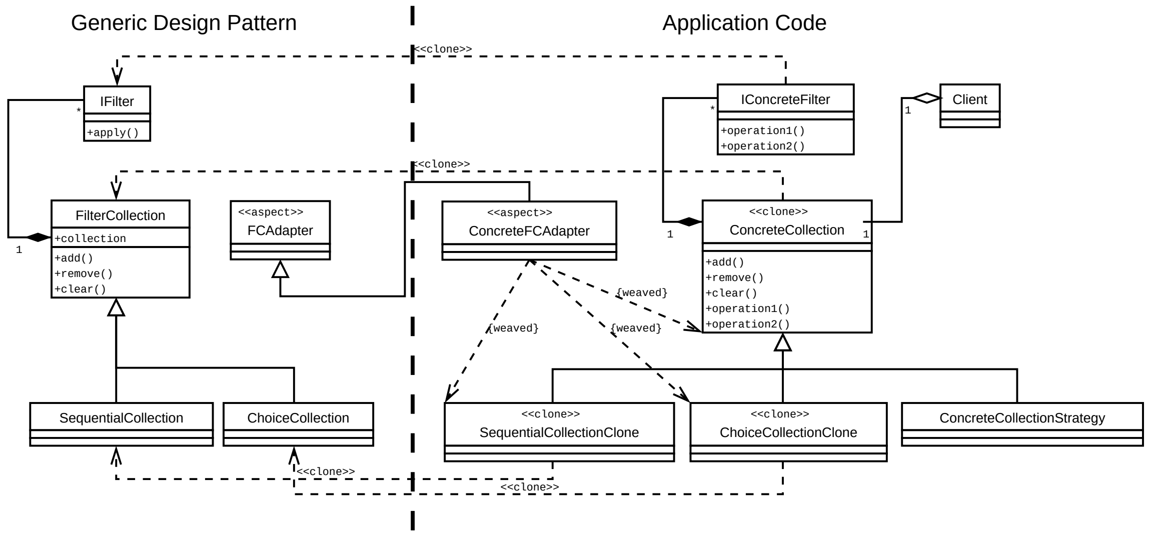

Figure 6: Diagram for the instantiation of the `FilterCollection` pattern, expressed using aspects and cloning with renaming.

```
        IFilter f = (IFilter) i.next();
        f.apply();
      } catch (Throwable e) {
        // Continue with next
      }
    }
  }
```

On the other hand, the `ChoiceCollection` tries to apply *one* filter in the collection. It uses exceptions to determine if a filter failed, and stops as soon as the application of a filter succeeded:

```
  public void apply() {
    for(Iterator i = _filters.iterator();
        i.hasNext();) {
      try {
        IFilter f = (IFilter) i.next();
        f.apply();
        break; // Try to apply once
      } catch (Throwable e) {
        // Try next
      }
    }
  }
```

Naturally, this hierarchy can be extended with more complex data-structures for collections and more complex strategies for filter applications, e.g., tree structured collections and prioritised strategies.

Inheritance is not an option for reusing the implementation in the `FilterCollection` hierarchy, since that would require the filter methods in the application to have the same method signature. By cloning the appropriate classes in this hierarchy we can reuse these classes in our application. All that is needed for this reuse is cloning of the hierarchy as follows:

```
  clone *FilterCollection+
     as *ConcreteFilterCollection
    with * * apply()
      as public void operation1()
    with * * apply()
      as public void operation2()
```

That is, clone each class in the hierarchy to a `Concrete` counterpart, and clone the `apply` method to each of the filter operations required by the `IConcreteFilter` interface of the application. In our concrete example of mail messages with processors and filters, we clone the `FilterCollection` hierarchy for each type of filter:

```
  clone *FilterCollection+
     as *ProcessorFilterCollection
   with * * apply()
```

```
      as public void process(Response req)

  clone *FilterCollection+
      as *MessageFilterCollection
    with * * apply()
      as public void apply(Message m)
    with * * apply()
      as public boolean validate(Message m)
         throws Exception e
```

With cloning we have achieved an application specific copy of the pattern hierarchy, but it requires further adaption. That is, the calls to the `apply` method within the clones of the `apply` method need to be forwarded to the concrete operations. For example, consider the result of cloning the `apply` method in the `SequenceMessageFilterCollection`:

```
  public void apply(Message msg) {
    for(Iterator i = _filters.iterator()
        ; i.hasNext(); ) {
      try {
        IFilter f = (IFilter) i.next();
        f.apply();
      } catch (Throwable e) { }
    }
  }
```

While the *signature* of the method has been adapted, the call to apply in the method is not. To adapt this call to the proper one, we use a *generic* aspect that adapts clones of `FilterCollection` to the specific context of there use. Figure 7 defines this aspect. The `callFilterMethodOnCollection` pointcut intercepts calls to methods targeting `IFilter` objects and uses reflection to store the method signature. Subsequent calls to `apply` *within* that methods are intercepted using the `callFilterMethod` pointcut and replaced with a call to the surrounding method.

Thus, we have encoded a general pattern as a class hierarchy and a generic aspect, that can be instantiated for use in specific applications by cloning. The advantage over the pure aspect approach of Hannemann and Kiczales [18] is that the resulting class hierarchy follows the normal OO design that one would use when manually encoding the pattern. For example, the filter collection that we have discussed here is similar to the Composite pattern by Hannemann and Kiczales. In their aspect solution, references to elements are stored in a `WeakHashMap` maintained in the aspect for all composites at once. Although a `WeakHashMap` discards its entries as soon as its key is discarded by the garbage collector, this does not work in the case of a circular references (e.g., a link from a file to its parent directory). Thus the aspect solution requires taking special care of this situation, whereas our implementation follows the conventional OO design, with the associated expected behaviour.

```
public aspect FilterCollectionAdapter {
  public void IFilter.apply() { }

  pointcut callFilterMethodOnCollection() :
    execution(* *.*(..))
    && target(IFilter);

  pointcut callFilterMethod(IFilter obj) :
    cflowbelow(callFilterMethodOnCollection())
    && call(* *.apply(..))
    && target(obj)
    && this(IFilter);

  private Object[] _args;
  private Class[] _argTypes;
  private String _methodName;

  before() : callFilterMethodOnCollection() {
      JoinPoint jp = thisJoinPoint;
      CodeSignature sig =
        (CodeSignature) jp.getSignature();
      // Save information as it is needed
      // later on
      _args = jp.getArgs();
      _methodName =
        jp.getSignature().getName();
      _argTypes = sig.getParameterTypes();
    }

  before(IFilter obj) : callFilterMethod(obj){
      JoinPoint jp = thisJoinPoint;
      Class cls = obj.getClass();
      Method method =
        cls.getMethod(_methodName, _argTypes);
      Object res = method.invoke(obj, _args);
    }
}
```

Figure 7: Generic aspect to adapt filter collection to concrete filter collection.

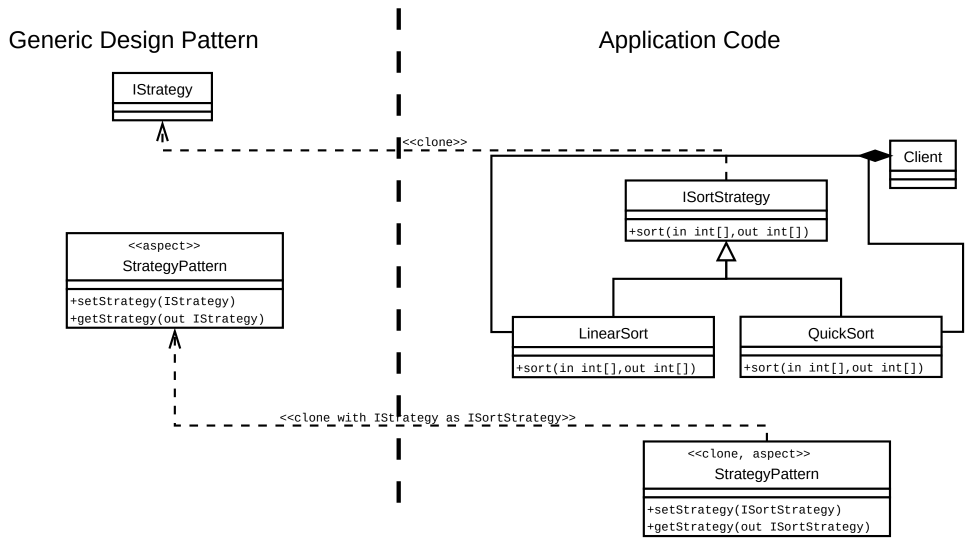
Generic Design Pattern
Application Code
IStrategy
<<clone>>
Client
ISortStrategy
+sort(in int[],out int[])
<<aspect>>
StrategyPattern
+setStrategy(IStrategy)
+getStrategy(out IStrategy)
LinearSort
+sort(in int[],out int[])
QuickSort
+sort(in int[],out int[])
<<clone with IStrategy as ISortStrategy>>
<<clone, aspect>>
StrategyPattern
+setStrategy(ISortStrategy)
+getStrategy(out ISortStrategy)

Figure 8: Diagram for the instantiation of the the `Strategy` pattern using aspects, cloning and renaming.

3.5 Strategy Pattern

According to [18], the `Strategy` pattern [15] may also be implemented for reuse with aspects. The implementation has two parts. It consists of an abstract interface `IStrategy` which all future strategies must implement, and an aspect called `StrategyPattern` which provides a small implementation of a strategy selector, as an intertype declaration. See the left hand side of Figure 8. The pattern is instantiated by extending `IStrategy` with the new operations for the particular algorithm in the instantiation. The aspect is then woven into the class which is supposed to hold the strategy.

This approach has an unfortunate drawback. All instantiations of the pattern will provide an interface which accepts any instance of `IStrategy`. Casting is performed to convert from from the `IStrategy` to the actual strategy interface being used in a given instantiation. This is obviously not type safe: the type system allows all strategies to be interchangeable, but doing this will result in a `ClassCastException`s at runtime.

We would like our instantiation of the `Strategy` pattern to only allow strategies suitable in our context. In fact, we want to reuse the pattern without inheriting from `IStrategy`. Cloning allows us to do this. The right hand side of Figure 8 shows how the generic aspect is cloned then extended using inheritance. The cloning declaration

```
clone StrategyPattern as StrategyPatternClone
      with IStrategy as ISortStrategy;
```

restricts the cloned aspect to only accept objects of the type ISortStrategy, thus solving our typing problem.

The client will set up his strategy as follows:

```
LinearSort linear = new LinearSort();
QuickSort quick = new QuickSort();

if (...) {
  SortStrategyPatternClone.aspectOf().
  setStrategy(linear);
} else {
  SortStrategyPatternClone.aspectOf().
  setStrategy(quick);
}
```

And later on, use it in the following manner:

```
int[] numbers = {5, 3, 2, 67, 42, 13};
ISortStrategy strat =
  SortStrategyPatternClone.aspectOf().
  getStrategy();
numbers = strat.sort(numbers);
```

3.6 Discussion

The `StrategyPattern` shows clearly that there are instances when inheritance is an inappropriate

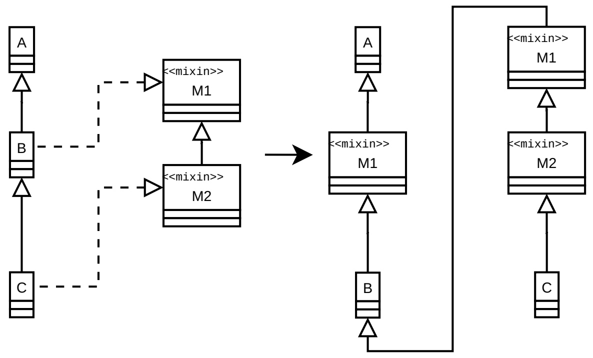

Figure 9: Mixin inheritance

technique for code reuse, because it tangles subtyping and composition. In many of these cases, the desired effect is exactly that which is provided by cloning.

A major drawback of current implementations of design pattern libraries (such as Hannemann and Kiczales [18]) is that they are sometimes too generic. The use of a design pattern should not force users to write type unsafe code or make concessions to their application design. Our solution is both generic (the template code) and specific (the cloned and renamed instantiation). This makes the code easier to understand and maintain, exactly what design patterns are aimed for.

4 Implementing Mixins

The literature is rife with descriptions of mixin classes (or just mixins). Its first appearance is often attributed to the Lisp community, in particular the Flavors system [27] and the mixin idiom in CLOS [5]. Later work has documented similar ideas expressed for Modula-3 [8], Java [2, 14] and Scala [28], to mention some. Limberghen and Mens [22] give another, more formal treatment of the topic.

In this paper, we mostly adopt the concept in the way it is described in [8]. A mixin is an abstract subclass which may be attached to a superclass, with the purpose of creating a family of modified classes. The attachment happens when the superclass is extended, and is controlled by the subclass. Figure 9 illustrates the usage. Class `B` (the client) extends `A` (the superclass) and attaches `M1`

```
public abstract aspect Border {
  public interface IBorder {};
  public void IBorder.paintBorder()
  { ... }
}

public aspect MixMessageWithBorder
  extends Border {
  declare parents: BorderMessage
    implements IBorder;
}

public class BorderMessage extends Message {
  public void paint()
  { paintBorder(); super.paint(); }
}
```

Figure 10: The Border mixin expressed as an aspect.

(the mixin) at the same time. A similar situation holds for `C`, which attaches both `M2` and `M1` to `B` at the time of extension.

The result of mixing is always a linear inheritance chain, as shown at the right hand side of the arrow in Figure 9. The linearisation is a central property in the mixin definition from [8], that we also adopted. This particular feature has attracted some critique [22].

In some languages without specific mixin constructs, idioms have evolved to capture the concept. See [32] for the C++ idiom as an example. Attempts at capturing mixins in AspectJ have been done [30]. The code in Figure 10 demonstrates this technique. Compared to full-blown mixins, the AspectJ idiom suffers from a few problems:

Name clashes: Signature conflicts are not allowed, i.e. two mixins cannot both introduce a method with the same signature, or a field with the same name.

Referring to Figure 10, we have the problem that `IBorder.paintBorder` cannot be named `IBorder.paint`. If it was, it could not be introduced into the `BorderMessage` class, as it would conflict with the `paint` method already defined there.

Inheritance: The introduced interface is not inserted as a parent of `BorderMessage`, nor as a subclass of Message. Name clashes notwithstanding, the lack of inheritance prohibits methods in the mixin from wrapping superclass functionality by the use of super.

Mixin layers: Because of the name clash issue, mixin layers [32] are difficult to implement, and requires careful pre-planning. When composing the layers, names are not allowed to collide. This places a severe restriction on the names available at the various layers in the mixin code.

Syntax: The syntactical verbosity is a problem both for human readers and tools. A mixin is not an aspect. Aspects are separate, crosscutting concerns which may occur at all abstraction levels from expressions to methods and classes, while mixins are about composition at the class level. In Figure 10, the mixin had to be embedded into an aspect, and may therefore easily be confused as such. Furthermore, any one mixin must be scattered into two aspects: the mixin definition itself, Border, and its application MixMessageWithBorder. The latter, although small, must be manually rewritten from scratch for each application of the mixin and must be kept in sync with its participants, BorderMessage and IBorder.

The mixin style supported by AspectJ is strikingly similar to traits, as described in [29]. Refer to Section 5 for more details.

4.1 Mixins using Clone and Rename

In [8], the authors present a generic recipe for implementing mixins in languages which do not directly support it: "the effect of a mixin can be achieved by explicitly creating subclasses and copying the mixin code into the subclass, preventing code sharing and abstraction".

Using the clone operator, we can eliminate the manual code copying, and using the concept of open classes [10], we can insert the copied class into the inheritance hierarchy. All we have to add is linearisation. Unfortunately, this is cumbersome to do manually. Also, the `declare parents` construct in AspectJ does not allow you to change the superclass of a class unless the new superclass is a subtype of the old one. In the case of mixins, this restriction must be lifted, as we do not want to manifestly fix the typing relation at mixin implementation time.

In order to investigate cloning plus open classes as implementation primitives for mixins, we have made a small syntactical extension to Java that makes mixins first class entities. Figure 11 details its syntax. The extension also embodies the linearisation algorithm that is characteristic to mix-

```
MixinDec      ::= mixin Identifier Extends?
Extends       ::= extends MixinName
MixinClassDec ::= ClassModifiers
                  class Identifier
                  TypeParams? Super? Mixins
                  Interfaces? ClassBody
Mixins        ::= mixes MixinName+
```

Figure 11: Syntax for the mixin extension. *MixinName* is a variant of *Identifier* which has been disambiguated by the Java/AspectJ type system. *Super*, *ClassModifiers*, *Interfaces* and *ClassBody* are identical to the Java Language Specification.

```
mixin Border {
  private void paintPreBorder() { ... }
  private void paintPostBorder() { ... }
  public void paint()
  { paintPreBorder(); super.paint();
    paintPostBorder(); }
}

mixin ShadowedBorder extends Border {
  private void paintPreShadow() { ... }
  private void paintPostShadow() { ... }
  public void paint()
  { paintPreShadow(); super.paint();
    paintPostShadow(); }
}

mixin NoisyComponent {
  private void emitSound() { ... }
  public void paint()
  { super.paint(); emitSound(); }
}

class FancyMessage extends Message
  mixes ShadowedBorder, NoisyComponent {
  FancyMessage(String m) { setMessage(m); }
}
```

Figure 12: Example of mixin inheritance

```
abstract class Border {
  public void paint()
  { ... ; super.paint(); ... }
}

abstract class ShadowedBorder
  extends Border { ... }

abstract class NoisyComponent { ... }

clone ShadowedBorder- as FancyMessage*;
clone NoisyComponent as FancyMessage*;

aspect MixBorderWithBorderedMessage {
  declare parents:
    FancyMessageBorder extends Component;
  declare parents:  FancyMessageNoisyComponent
      extends FancyMessageBorder;
  declare parents: FancyMessageBorder
      extends FancyMessageNoisyComponent;
}
```

Figure 13: A sketch of how mixins are conceptually realized using cloning and open classes. Note that due to the unresolved `super` in `Border.paint` and `declare parents` inability to change the supertype of a class, this example will not compile in AspectJ.

ins. We illustrate its usage by the classical border example, shown in Figure 12. Given class `Message` and `Component` as already described in Figure 3, we want to extend `Message` with a fancy border and some captivating sound. The declaration of `FancyMessage` in this example will, after linearisation, result in the inheritance chain `FancyMessage` → `NoisyComponent` → `ShadowedBorder` → `Border` → `Message` → `Component`.

(As our goal is to show how cloning enables mixins, we will not argue why mixins should be preferred over inserting `Message` into a `BorderContainer` or adding border painting as an `around` advice.)

The resolution of this mixin inheritance is pretty straightforward. A sketch of its elementary units of operation is shown in Figure 13. First, we view each mixin as an abstract class. It is important to note that a mixin cannot be compiled separately, since it may freely rely on fields and methods in its superclass. These undefined references will not be fixed until the mixin is attached to a superclass. This property carries over to the abstract class view as well.

Second, all mixins (still in the form of abstract classes) are cloned. This is crucial. The cloning is the enabling operator that allows us to modify the inheritance properties of the mixin so that a mixin may be mixed into several hierarchies independently. Without cloning, a mixin could only be attached once.

Third, a slightly relaxed `declare parents` operator is applied to connect the cloned mixins with the super- and subclasses. The AspectJ `declare parents` does not allow arbitrary inheritance modification, but this rule turns out to be necessary in our case.

4.2 Discussion

If we compare our mixin implementation with mixins as expressed in AspectJ, we can note the following:

Name clashes: Because of the linearisation, name clashes are no longer a problem. If a mixin provides an identical signature with its superclass, subclass or another mixin, it will be resolved using the single dispatch mechanism provided by the Java runtime.

Inheritance: As mixins are now inserted into the inheritance hierarchy, we can exploit this in our border examples. In `Border.paint` in Figure 12, we first draw part of the border, have the superclass draw itself, then draw the rest of the border.

Mixin layers: Our current prototype implementation does not directly support multiple levels of mixins, i.e. you cannot declare `mixin A mixes B`, thus mixin layers are not significantly easier to implement than using the open class mechanism in AspectJ. Extending our prototype to include this feature would be straightforward. Only the linearisation code must be extended.

Syntax: With the concrete mixin syntax we have suggested, the syntactical representation is straightforward and readable. Even without it (viewing mixins as abstract classes), there is a slight improvement, as mixins will appear as abstract classes, and not embedded into aspects.

The user of mixins must also be aware of a few issues:

Duck typing: [2] Our mixins are not separately compilable. They may be *partial*, that is, they contain references to methods and fields meant to be provided by the superclass they will be attached to. These references will be resolved and type checked at mixin time. This makes them very flexible—they are applicable wherever their open references are resolvable—but also slightly foreign to the Java paradigm.

We experimented with adding a `requires` clause to our mixins. This would dictate an interface that must be fulfilled by the parent for the mixin to be applied. It would eliminate the reliance on duck typing, but require superclasses to anticipate their mixins. Alternatively, AspectJ open classes could be used to introduce new interfaces on the superclass post-facto, thus providing the mixin developer with stricter control of where his mixins may be applied safely, but this places a questionable burden on the mixin user.

Data members: When a mixin introduces its own fields, these may unintentionally overlap with those of the superclass when there is a naming conflict. The Java semantics resolves these conflicts by having subclass fields shadow those of the superclass. This problem with mixins has been attacked in [29].

Constructor chains: A problem occurs when a mixin is applied to a superclass without a default constructor. Either the mixin developer must have anticipated this by providing the mixin with a suitable constructor which forwards to the correct superclass constructor, or an appropriate "glue"-constructor may be introduced into the mixin using local aspects at the time of mixing.

In summary, our prototype implementation shows that mixins may be realized using only cloning and slightly relaxed intertype declarations.

[2]From the proverb "if it walks like a duck and talks like a duck, it must be a duck".

5 Related Work

5.1 Related Language Features for Reuse

Here we discuss related language features which offer solutions to the code reuse problem, and contrast them to the use of cloning.

Aspects [20] modify existing class hierarchies, preventing generic reuse of woven classes in multiple contexts within the same program.

Mixins [8] allow reusable behaviour, but do not allow separate implementation of cross-cutting concerns. Mixins have been described in many OO languages, among them Smalltalk [8], Jam for Java [2], CLOS for Lisp [5] and the Scala language [28].

Traits differ from mixins in that they do not have state, and they do not require linearisation. "A trait is a group of pure methods that serves as a building block for classes and is a primitive unit for code reuse" [29]. They have a *flattening property*, which states that "the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all the non-overridden methods of the traits". Traits have been explored in many OO-languages, among them Smalltalk [12, 29], and Scala [28]. This can already be expressed in AspectJ using intertype declarations in the style of Figure 10. No cloning is necessary.

Templates are available in C++ [33] and for Haskell in TemplateHaskell [31]. They provide a powerful and flexible technique for code reuse decoupled from inheritance, but require that the code is expressed as templates from the beginning. With careful use of cloning and renaming, we can get some of the same benefits of templates. Virtual classes [24] is an alternative technique for describing generics.

Inheritance [34] is the fundamental technique for code reuse on which all object-oriented language have been built. The exact details of what inheritance is, varies between languages, but is usually taken to mean a combination of code reuse, subtyping (the *is-a* relationship) and dynamic dispatch. Our *clone* operator is only about code reuse, decoupled from the other elements of inheritance (dispatch techniques, typing relationship).

Open classes [10] or intertype declarations in AspectJ allow methods and fields to be added into existing classes without breaking encapsulation. In Java, this means that newly inserted methods at best only have friendly access, and cannot see the private parts of a class. Open classes may only modify existing classes in-place. A system which brings this to Java, along with some forms of component composition, is Jiazzi [25].

At their most general, meta object protocols [19] make the entire language open for programming, and blurs the distinction between runtime and compilation time. They allow arbitrary meta programs, written in an object-oriented style, to be applied to a program both at compile-time and runtime. Our cloning operator is equal to copying (cloning) a meta object representing a class, method or field.

Module systems have been described for imperative (Modula-2), object-oriented (e.g., Java packages) and functional [21, 23] languages. Visible, named entities from a module may be imported from one module to another, and renamed in the process. This creates an alias for the code in the original module. Unlike cloning, code modification performed on the new name will affect the original (only) definition, as the name is merely an alias. This prohibits the use of aspects for context-specific adaptation. Coupling module imports with a safe solution to cross-package cloning would provide module systems with per-context aspect weaving.

Higher-order hierarchies [13] is a language feature in the *gbeta* language, proposed as a solution to improve reuse by minimising the need for boilerplates (copy-then-modify reuse). It allows the extension of class hierarchies into new, modified hierarchies. The author points out that unlike aspects, higher-order hierarchies allow both the new and old hierarchies to co-exist side-by-side in a given system, and argues that this is preferable to the in-place modification required by aspects. Using cloning we can create a copy of a class hierarchy, then adapt it using aspects, offering similar functionality to higher-order hierarchies. The major difference is that in higher-order hierarchies, the new, extended hierarchy enjoys a well-defined typing relation with the old, whereas in our approach there is none. This allows pure code reuse decoupled from inheritance.

Most object-oriented (as well as many imperative and functional) languages allow the expression of callbacks, closures, delegates and aggregation, either as idioms or through dedicated language constructs. In languages where callbacks, closures and delegates are named entities, they may be cloned and adapted, provided visibility is respected. New callbacks may be exposed by adapted code; advice may be used to expose new variation points in a class after it has been cloned. In some situations, when aggregation and delegation is about flexibly adding new behaviour to a class, mixins (through cloning and open classes) may be substituted. This allows the preservation of object identity.

5.2 Related Design Techniques for Reuse

Design patterns are elements of reusable engineering knowledge. They are are recipes for a given class of recurring problems and their solution, stated in general terms. A design pattern must be "instantiated" by the developer, in the form of a concrete implementation.

Component engineering is about constructing reusable elements at a substantially coarser granularity than the class-level. For the reasons of encapsulation discussed previously, we do not believe cloning and renaming make much sense at this level of abstraction.

Frameworks are object-oriented class hierarchies constructed using available language constructs, such as inheritance, aggregation, and even mixins, traits, templates, where available. When the available language features are not expressive enough, idioms and design patterns are resorted to.

Invasive software composition [4] is about composing software from reusable components, and adapting them at composition time. The starting point is a minimal set of program transformations which are used to build composition operator libraries. In turn, these parameterise, extend, connect, mediate, and aspect-weave components as part of the composition process. In the more general context, our approach can be viewed as an instance of this general technique.

6 Discussion

Cloning of packages, classes, methods, fields can be expressed in many languages offering meta programming facilities [5] or meta object protocols [9, 35]. Program transformation systems [36, 11] also provide this capability. While these systems are extremely expressive, they are also highly complex and require both discipline and experience to use. It is our belief that their complexity can be tamed and distilled into useful, high-level, declarative language constructs. In the case of cloning, we view these technologies as competing alternatives for expressing its implementation. Our prototype is implemented entirely in the Stratego program transformation system [36].

Cloning entire packages is problem-free, as they are self-contained units with respect to visibility. Cloning classes across package (or module) barriers is desirable, but without runtime support, this is difficult in languages providing encapsulation features. Unless a class was designed to be cloned, by not referencing any non-public parts of its package, it cannot safely be cloned into a new package. In a sense, mixins are a special case, since they by nature should be designed for cloning. Several trade-offs to tackle this problem are possible. The simplest is to disallow cross-package cloning entirely. A more flexible alternative would be to make the clone a "foreign" citizen in the new package. To its clients, the clone appears to reside in the new package, but retains only privileges from its old package.

With the relaxed `declare parents` operator (see Section 4), we can insert mixins into existing inheritance chains, between any two classes. This differs from pure intertype declarations on two areas: (1) the introduced code may be expressed and treated as a composable class – a mixin, and (2) the methods in the mixin may augment the methods in their parent class by use of `super`. For our bordered message example in Figure 12, a similar effect to `super` in the mixin `paint` may be attained by implementing the `paint` method as an `around` advice, but this approach is verbose and requires extra care. The advice should be applied on a per-context basis. We cannot weave the border `around` advice directly on `Message`. That would turn all `Message`s into `FancyMessage`s, clearly not what we want. Using intertype declarations, we may work around this by adding a member flag to the `Message` class which when set, will have the advice draw the border, giving us a per-object granularity at the cost of a slight runtime overhead. It is our opinion that in this case, it is more of an implementation trick to emulate mixins than a clean design.

Language constructs for capturing design patterns [6] and reusable libraries of design patterns [1] are appealing because they make the patterns into concrete, reusable entities, and also allow concise embeddings of the patterns into the code. An minor annoyance identified in [1] is that design patterns expressed as reusable code sometimes have overly generic method names, such as `anOperation`. When instantiating the design pattern, the name must be kept. Clearly, this can easily be avoided when the pattern is instantiated using cloning and renaming.

An obvious critique against the clone operator is that it actually duplicates code. The code for every named identifier it is applied to is in fact copied. Unlike classical code duplication, this kind of code duplication is only taxing on the runtime environment, not the developer or program designer. In theory, the duplication could be minimised in a sufficiently advanced runtime, by employing a copy-on-adapt scheme: the copying will actually only occur when the cloned version is used and modified. Furthermore, only the actually modified parts (methods and fields inside classes) need to be cloned. Even without runtime support, the cost of cloning is not likely to be prohibitive. Most design techniques were invented exactly to avoid copy-then-modify, which cloning now provides as a basic language feature. While the gap in the design space filled by cloning is small enough for code duplication not to be a serious issue, we have argued that is is an important gap.

7 Conclusion and Further Work

We have detailed the *clone* operator, a small, declarative extension to the Java language which improves code reuse. We have discussed its implementation, how it can be combined with intertype declarations to implement mixins, how it combines with the aspect language AspectJ to allow per-context weaving, how it can be used as a primitive template mechanism and how it can be combined with aspects to implement reusable design patterns as code.

We have related our work to alternative techniques for code reuse and adaptation for object-oriented languages and argued that the concept of cloning is either already inherent in the solution, or would be complementary and beneficial.

We clearly rely on mechanisms other than cloning to perform code adaptation. In this article, we have used aspects and intertype declarations to declaratively express these adapting transformations. While we have shown that there are useful and interesting cases for this approach, there is no hiding the fact that aspects have a limited capability for program transformation. In future work, we seek to uncover alternative adaptation mechanisms which preserve the declarative nature of aspects and intertype declarations, but combine better with the cloning operator.

Acknowledgements: We thank the Research Council of Norway for funding authors Bagge and Kalleberg, and extend our thanks to Karina Olmos for constructive feedback on early drafts of this report.

References

[1] E. Agerbo and A. Cornils. How to preserve the benefits of design patterns. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 134–143. ACM Press, 1998.

[2] D. Ancona, G. Lagorio, and E. Zucca. Jam—designing a Java extension with mixins. *ACM Trans. Program. Lang. Syst.*, 25(5):641–712, 2003.

[3] AspectJ homepage. http://eclipse.org/aspectj/.

[4] U. Assmann. *Invasive Software Composition.* Springer-Verlag New York, Inc., 2003.

[5] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kicsales, and D. A. Moon. Common LISP object system specification X3J13 document 88-002R. *SIGPLAN Not.*, 23(SI):1–143, 1988.

[6] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[7] G. Bracha. *Generics in the Java Programming Language.* Sun Microsystems, Inc., 2004.

[8] G. Bracha and W. 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. ACM Press, 1990.

[9] S. Chiba. A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, Oct. 1995.

[10] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.

[11] J. R. Cordy, C. D. Halpern, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. In *Proceedings of the IEEE 1988 International Conference on Computer Languages*, pages 280–285, October 1988.

[12] G. Curry, L. Baer, D. Lipkie, and B. Lee. Traits: An approach to multiple-inheritance subclassing. In *Proceedings of the SIGOA conference on Office information systems*, pages 1–9, 1982.

[13] E. Ernst. Higher-order hierarchies. In L. Cardelli, editor, *Proceedings ECOOP 2003*, LNCS 2743, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[14] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM Press, 1998.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, Reading, Massachusetts, USA, 1995.

[16] J. Gil and D. H. Lorenz. Design patterns vs. language design. In *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology*, pages 108–111. Springer-Verlag, 1998.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Sun Microsystems, Inc., second edition, 2000.

[18] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[19] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA, 1991.

[20] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[21] X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[22] M. V. Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object Oriented Systems*, 3:1–30, 1996.

[23] D. MacQueen. Modules for standard ml. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM Press.

[24] O. L. Madsen and B. Moller-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. ACM Press, 1989.

[25] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: new-age components for old-fasioned java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 211–222, New York, NY, USA, 2001. ACM Press.

[26] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99. ACM Press, 2003.

[27] D. A. Moon. Object-oriented programming with flavors. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–8. ACM Press, 1986.

[28] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Michelou, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The Scala language specification. Technical report, EPFL, Switzerland, March 2005.

[29] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP*, pages 248–274, 2003.

[30] A. Schmidmeier, S. Hanenberg, and R. Unland. Implementing known concepts in AspectJ. In *3rd Workshop on Aspect-Oriented Software Development of the SIG Object-Oriented Software Development of the German Informatics Society*, Essen, Germany, 2003.

[31] T. Sheard and S. P. Jones. Template metaprogramming for Haskell. In *Haskell '02: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM Press, 2002.

[32] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[33] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, USA, third edition, 1997.

[34] A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[35] M. Tatsubori, S. Chiba, K. Itano, and M.-O. Killijian. OpenJava: a class-based macro system for Java. In *Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering*, pages 117–133. Springer-Verlag, 2000.

[36] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.