How generic language extensions enable "open–world" design in Java

Marco Nissen, Karsten Weihe

MPI–I–1999–004 September 1999

Author's Address

Marco Nissen
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken
marco@mpi-sb.mpg.de
http://www.mpi-sb.mpg.de/~marco/

Karsten Weihe
Universität Konstanz
Fakultät für Mathematik und Informatik, Fach 188
78457 Konstanz
Karsten.Weihe@uni-konstanz.de
http://www.fmi.uni-konstanz.de/~weihe/

Abstract

By *open–world design* we mean that collaborating classes are so loosely coupled that changes in one class do not propagate to the other classes, and single classes can be isolated and integrated in other contexts. Of course, this is what maintainability and reusability is all about.

In the paper, we will demonstrate that in Java even an open–world design of mere attribute access can only be achieved if static safety is sacrificed, and that this conflict is unresolvable *even if the attribute type is fixed*. With generic language extensions such as GJ, which is a generic extension of Java, it is possible to combine static type safety and open–world design. As a consequence, genericity should be viewed as a first–class design feature, because generic language features are preferably applied in many situations in which object–orientedness seems appropriate.

We chose Java as the base of the discussion because Java is commonly known and several advanced features of Java aim at a loose coupling of classes. In particular, the paper is intended to make a strong point in favor of generic extensions of Java.

Keywords

Java, GJ, Generic Programming, Data Accessor, Algorithm Engineering

Contents

1 Introduction

If collaborating classes are strongly coupled with each other, none of them can be modified in isolation or incorporated into a different context. In other words, maintenance and reuse are seriously limited. Basically, collaboration of two classes $\mathcal{A}$ and $\mathcal{B}$ means that objects of class $\mathcal{A}$ access the methods of objects of class $\mathcal{B}$ (often also vice versa).

In many software designs, method access is the main reason for strong coupling: as long as objects of $\mathcal{B}$ are merely "passed on" by objects of $\mathcal{A}$ and no method of $\mathcal{B}$ is called inside $\mathcal{A}$, the implementation of $\mathcal{A}$ may treat $\mathcal{B}$ objects as more or less anonymous (*e.g.* as objects of type *Object* in Java), so $\mathcal{A}$ and $\mathcal{B}$ are particularly loosely coupled[1].

Design concepts based on event sending (like in *JavaBeans* [6]) provide an alternative to direct method access, which allows a weaker coupling. However, such a design is certainly not a feasible alternative under all circumstances. Hence, the problem of flexible, yet safe method access is still important. By *safe* we mean *statically safe* in the first place, that is, if a method of an object is called in some piece of code,[2] a static analysis of this piece of code at compile time is able to determine whether this object offers the required method and the signature[3] of this method is also as required.

There has been a long–standing debate in various scientific and other communities whether *static* safety at compile time is important or *dynamic* checks at run time would be sufficient. In fact, various languages (notably *Smalltalk*) do not offer static safety at all. It is our feeling that there is no general answer to this question: static safety is highly desirable in some situations, and not of any use in other situations. We will analyze this problem in Section 2.3. It will turn out that the differences between these two kinds of situations are rather subtle.

The discussion in this paper will be along the lines of a concrete, step–by–step

[1]Sectionn 2.2 explains loose coupling in greater detail.

[2]We use the unspecific term "piece of code" here and at other points to avoid terms that are ambiguous ("component," "module," etc.) or whose meanings are too specific for our purposes ("subroutine," "function," "method," etc.).

[3]The signature of a method comprises its name, list of argument types, return type, and the exceptions thrown by this method. The signature of a class or interface is determined by the signatures of its public methods.

case study, which is introduced in Section 2. This case study is taken from a realm that is quite common in Java programming: the processing of visual data.

Section 2 starts with a case study about attribute access in Java and the following sections describe the disadvantages of the first solution. Sections 2.2 and 2.3 explain in detail what is meant by loose coupling and static type safety. Sections 2.4–2.6 will demonstrate that even the advanced features, which are applicable for attribute access in Java do not sufficiently support these goals (not even in case the type of the attribute is fixed). Section 2.7 demonstrates the problems that are introduced when the attribute type is made variable in an algorithm. Section 2.8 shows how loose coupling and static type safety may be resolved through a parametrically polymorphic design, which is not possible in pure Java, but in generic language extensions such as GJ [2] (see [1] for a relatively recent survey of generic Java extensions).

From the reader we assume Java or C++ literacy and familiarity with object–oriented programming concepts such as classes and inheritance. The appendices at the end of the article briefly introduce specific Java and GJ concepts (enumerations, inner classes, reflection, and parametric polymorphism), which do not have exact counterparts in other object–oriented languages.

An extended version of the code examples quoted in this manuscript can be found at

`http://www.mpi-sb.mpg.de/~marco/OpenWorld/`

2 Simple Case Study

Our running example is a simple shape–oriented image–processing algorithm, which accesses abstract attributes of two–dimensional geometric shape objects via a pair of get/set methods.

For example, the color and texture of a (monochromatic) geometric object are two typical abstract attributes. If the borderline and the interior of a shape may have different colors, each geometric object has two color attributes: borderline color and interior color.

In this article, an (abstract) attribute of a class or interface $\mathcal{A}$ . . .

- . . . is a conceptual, abstract entity.
- . . . is associated with objects of class / interface $\mathcal{A}$.
- . . . is publicly accessible.
- . . . has a certain data type (primitive or class type).

Abstract means that its implementation is left open. The usual way of attaching an attribute to a class $\mathcal{A}$ is to make it a private data member of $\mathcal{A}$ and to add access methods for this member to the public part of $\mathcal{A}$ (often, the word *attribute* is exclusively used to refer to data members of classes). However, other ways of realizing an abstract attribute are sometimes more suitable. We will come back to this point later on (Section 2.2).

It goes without saying that this simple example is only a representative of more complex scenarios, in which the problems discussed here are even more urgent.

2.1 Straightforward Implementation

For simplicity, we assume in our running example that geometric objects are purely monochromatic. However, we do not regard the color of an object as a single attribute, but as a composition of three attributes: red, green, and blue, according to the *RGB* encoding scheme. In other words, the color of a geometric object is represented by a triple (red,green,blue) of `Double` objects, each in the range from 0 to 1. Then (1,0,0) means plain red,(0,1,0) is green, and (0,0,1) is blue. The code (0,0,0) stands for black, and (1,1,1) for white, and so on. This encoding scheme for colors is also used in the Java color model, which is realized by class `java.awt.image.ColorModel`.

Two–dimensional geometric shapes may then be represented by an interface[1] like the following, which we name `GeomShape2D`.

For technical reasons, which will become evident later on, we will not use the primitive type `double`, but the corresponding wrapper class `java.lang.Double`:

```
public interface GeomShape2D {
   public Double getRed ();
   public Double getGreen ();
   public Double getBlue ();
   public void setRed (Double r);
   public void setGreen (Double g);
   public void setBlue (Double b);

   // Further general methods for arbitrary two-dimensional
   // shapes, e.g. for shifting or rotating a shape object
}
```

We have added UML diagrams to the source code examples to improve the understanding of the text.

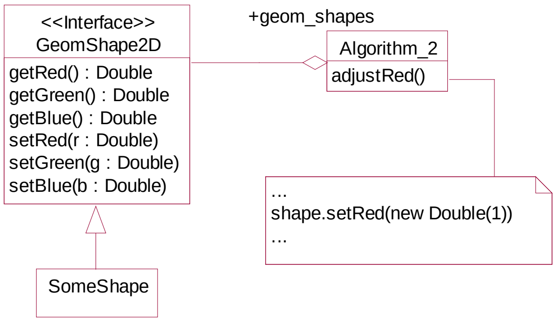
UML means unified modelling language and can be used to describe e.g. class hierarchy and dependecies. For example, in figure 2.1, class SomeShape is derived from the interface GeomShape2D. The line between Algorithm_2 and GeomShape2D means that there is a relation. Here, the geometric shapes of the algorithm are weakly associated with the algorithm, i.e. the algorithm holds a set of references. We omitted the number of references, but in the whole paper, geom_shapes always represents a set of references, and all other relations are single references (like myColor in figure 2.2). The method adjustRed() is always annotated with a small box containing a fragment from its implementation. Sometimes the signatures of the methods are given as well - method (parameter : Integer) : Double means that this method takes a single parameter 'parameter' of type integer and returns an object of type Double.

Therefore we can describe figure 2.1 as follows: An algorithm class has a method adjustRed() which can be called to do some modification on a set of geometric two-dimensional shapes. A concrete shape (like a circle, line or square) may be derived from the base class. The method adjustRed accesses the attribute values through the interface of each shape, i.e. by calling the methods provided by GeomShape2D.

For example, an image composed of two–dimensional shapes may be modeled as an object of class `java.util.Vector`, whose items shall be objects of

[1] The C++ literate reader may think of a pure abstract class.

Figure 2.1 UML-diagram for straightforward implementation



classes implementing interface `GeomShape2D`:

$$\boxed{\textbf{Vector}\ \text{geom\_shapes} = \textbf{new Vector}\ ();}$$

For the sake of argument, suppose we want to perform the following simple image processing operation: the RGB–values of each geometric object in `geom_shapes` are modified such that the sum of the three color values (the *brightness*) is not changed, but the contribution of red is increased by 10%[2]. The mathematical details of the algorithm are not relevant. The main point is the impression that even such a simple algorithm may result in a complex, error–prone implementation, which is hard to understand and even harder to maintain.

[2] If red already contributes more than 90% to the color of an object, we set the color of this object to plain red, that is (1,0,0).

```
public static void adjustRed (Enumeration enum, double percentage) {
  // See Appendix A on Enumerations for a brief introduction into this Java feature.
  while (enum.hasMoreElements ()) {
    Object obj = enum.nextElement ();
    GeomShape2D shape =(GeomShape2D)(obj);
    double red = shape.getRed ().doubleValue ();
    if (red > 1-percentage/100) {
      shape.setRed (new Double (1));
      shape.setGreen (new Double (0));
      shape.setBlue (new Double (0));
    } else {
      double new_red = red + percentage/100;
      double green = shape.getGreen ().doubleValue ();
      double blue = shape.getBlue ().doubleValue ();
      shape.setRed (new Double (new_red));
      shape.setGreen (new Double ((1-new_red) *(green/(green+blue))));
      shape.setBlue (new Double (1-new_red-green));
    }
  }
}
```

Clearly, unlike in C and C++, every function must be a method of some class. Let's assume the class of `adjustRed` is a mere container of geometric algorithms and called `GeomAlgorithms`. Then a call to `adjustRed` to increase the contribution of red by 10% looks like this:

```
GeomAlgorithms.adjustRed (geom_shapes.elements (), 10.0);
```

The key word `static` in the header of `adjustRed` allows a call to `adjustRed` without an actual object of class `GeomAlgorithms`, simply by qualifying the name of the class.

2.2 Goal 1: Loose Coupling

The fact that geometric objects are represented by interface `GeomShape2D`, and that the basic colors red, green, and blue are accessed by methods `getRed`, `setRed`, etc., is "hard–wired" in the algorithm `adjustRed`. Such a strong coupling of a piece of code with its context is generally undesirable:

1. It is very likely that future maintenance work will affect the protocol used to tie a piece of code together with its collaborators. It is even more likely that a new application context will impose a completely different protocol.

2. It is highly uneconomical and error–prone to implement non–trivial pieces of code repeatedly from scratch (or to revise them exhaustively) whenever

the context changes through maintenance work or the subroutine shall be applied in a different context.

The goal is to minimize strong coupling. A positive formulation is to design loosely coupled software.

To stick to our running example: for `adjustRed` this means it cannot be assumed that the underlying geometric shape class will always be named `GeomShape2D`, that the RGB values are always accessed through methods `get/setRed`, etc., and that these methods always have the same signatures as in `GeomShape2D`. Even a subroutine as simple as `adjustRed` is complex enough to make the adaptation to another "protocol" a potentially hazardous effort, because the "volatile" details are helplessly intermixed with the logic of the subroutine.

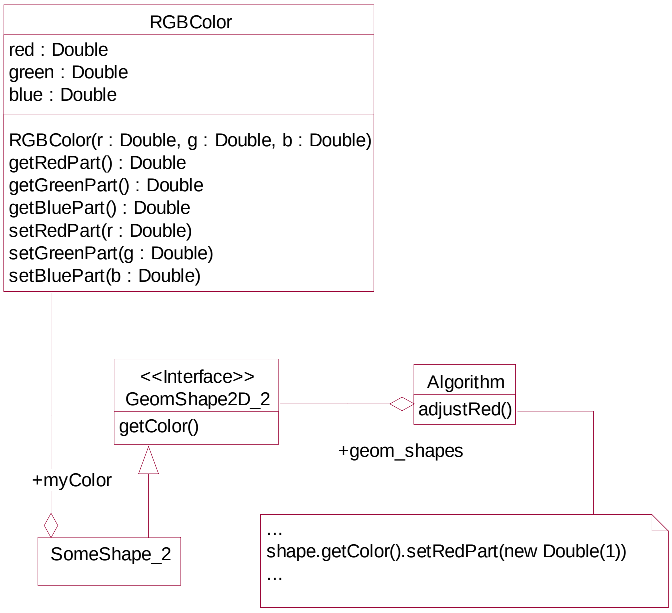
At first glance, this statement may look a bit too strong, because it seems that the details to be changed can be easily found in the code and modified by straightforward changes. However, the discussion of the following three variations will give an imagination how complex such a modification actually may be in practice. Each of these variations is realistic and has been found in projects, and adapting an algorithm such as `adjustRed` to any of them is by no means trivial.

2.2.1 First variation: combined attributes

In the first variation of `GeomShape2D`(*i.e.* interface $\texttt{GeomShape2D}_2$ below), the three basic colors are not realized as three mutually independent attributes of `GeomShape2D`, but form one attribute of type `RGBColor`, say, which comprises three `Double` values (see figure 2.2 for the corresponding UML diagram):

```
class RGBColor {
  Double red, green, blue;
  RGBColor (Double red, Double green, Double blue) {
    this.red = red;
    this.green = green;
    this.blue = blue;
  }
  public Double getRedPart () {
    return red; }
  public Double getGreenPart () {
    return green; }
  public Double getBluePart () {
    return blue; }
  public void setRedPart (Double red) {
    this.red = red; }
  public void setGreenPart (Double green) {
    this.green = green; }
  public void setBluePart (Double blue) {
    this.blue = blue; }
}
```

Figure 2.2 UML-diagram for first variation



The following variant of GeomShape2D represents geometric objects whose colors are implemented by class RGBColor:

```
public interface GeomShape2D_2 {
  public RGBColor getColor ();
  // Further general methods for arbitrary two-dimensional
  // shapes, e.g. for shifting or rotating a shape object
}
```

2.2.2 Second variation: separate attributes

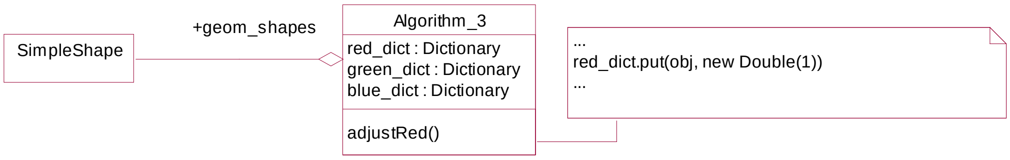
So far, we did not distinguish between abstract attributes and data members of classes, which are also often called attributes. However, an attribute is not necessarily a data member but, more generally, a *conceptual* entity related to a class, which has a designated type and associates a value of this type to every object of the class. An attribute may be *implemented* as a data member of this class (for example, accompanied by a pair of get/set methods as above). However, it may also be implemented completely differently, as we will see in the scenario that we are going to discuss next.

If an attribute of a class $\mathcal{A}$ is implemented as a *data member* of $\mathcal{A}$, it is permanently associated with $\mathcal{A}$. Sometimes it is useful or even necessary to attach an additional, *temporary*, attribute to existing objects of type $\mathcal{A}$. For instance, if geometric shapes are not designed to have a color, the objects in a collection of GeomShape2D (such as geom_shapes) cannot be assigned color values other than by storing all of these values in separate, additional data structures. For example, in Java the three additional RGB values could be stored in three separate dictionaries from java.util.Dictionary, which are realized by java.util.HashTable (see again figure 2.3 for the corresponding UML-diagram):

```
Dictionary red_dictionary   = new Hashtable ();
Dictionary green_dictionary = new Hashtable ();
Dictionary blue_dictionary  = new Hashtable ();
```

The formal type of items in dictionaries is Object, however, the corresponding implementation of adjustRed assumes that all objects in these three dictionaries are of subtype Double:

Figure 2.3 UML-diagram for second variation




```
public static adjustRed2(Enumeration enum, Dictionary red_dict,
    Dictionary green_dict, Dictionary blue_dict, double percentage) {
    while (enum.hasMoreElements ()) {
        Object obj = enum.nextElement ();
        double red = red_dict.get (obj).doubleValue ();
        if (red > 1-percentage/100) {
            red_dict.put (obj, new Double (1));
            green_dict.put (obj, new Double (0));
            blue_dict.put (obj, new Double (0));
        } else {
            // etc.
        }
    }
}
```

In other words, to set and retrieve a basic color of a geometric object `obj`, this object serves as the key to the methods `put` and `get` of `Dictionary`.

It is beyond the scope of this paper to discuss whether storing attributes in separate containers is a reasonable approach (see Section 2.2 of [7] for an in–depth discussion). Here we only note that designs like this actually appear in practice and have to be coped with.

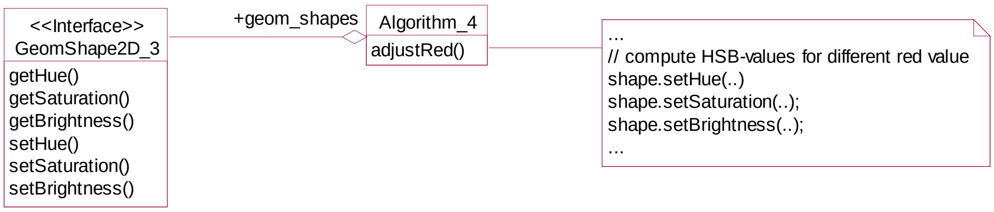
2.2.3 Third variation: different encoding

RGB is but one possible encoding scheme for colors. For example, it is also possible to express every color as a composition of a *hue* value, a *saturation* value, and a *brightness* value. This is the well–known *HSB* encoding scheme for colors. The RGB and HSB encoding schemes are equivalent in the sense that there are two algorithms that translate an RGB specification into an HSB specification and vice versa. The following, third version of `GeomShape2D` represents the color of a geometric object according to the HSB scheme (see figure 2.4 for the corresponding UML-diagram):

```
public interface GeomShape2D_3 {
    public Double getHue ();
    public Double getSaturation ();
    public Double getBrightness ();
    public void setHue (Double h);
    public void setSaturation (Double s);
    public void setBrightness (Double b);

    // Further general methods for arbitrary two-dimensional
    //shapes, e.g. for shifting or rotating a shape object
}
```

Figure 2.4 UML-diagram for third variation



Summary of Section 2.2

A feature of a class (even a feature as simple as a single attribute!) may appear in quite different ways in the class' interface. A true open–world design of a client must be able to cope with all of them.

2.3 Goal 2: Static Safety

This section is intended to clarify our viewpoint on static (type) safety. Static safety can be destroyed by an evaluation of run–time type information. For example, the down–cast in `adjustRed` from `Object` to `GeomShape2D` implicitly evaluates the run–time information whether or not the argument's class implements the interface `GeomShape2D`.[3] In Java, this information can be explicitly queried through operator `instanceof`. On the other hand, *reflection* (Appendix C) allows the run–time access to detailed information about the properties of an object's type.

[3]Casts in C and C++ are another example of implicit run–time type evaluation, which is even more dangerous, because here a type error does not raise an exception but may result in undefined behavior.

We do not regard *every* application of run–time type information as loss of static safety. In fact, we will distinguish between two fundamentally different *use scenarios* of run–time type information, which look very similar but have fundamentally different semantics. In the first use scenario, a maximum degree of reliability is achieved despite the fact that run–time type information is heavily incorporated. In contrast, the second use scenario reveals a serious gap in static safety.

Note that the implementation of `adjustRed` in Section 2.1 is not really complete, because a failure of the down–cast from `Object` to `GeomShape2D` is not caught. We will consider two different ways of catching such a failure. It will turn out that these two ways are quite representative for the two different use scenarios.

A brief sketch of the first variant:

```
public static void adjustRed3(Enumeration enum, double percentage) {
   while (enum.hasMoreElements ()) {
      Object obj = enum.nextElement ();
      if (obj instanceof GeomShape2D) {
         GeomShape2D shape =(GeomShape2D)(obj);
         // Proceed with the algorithm...
      } else {
         throw MyFavoriteException ();
      }
   }
}
```

And here is the second variant:

```
public static void adjustRed4 (Enumeration enum, double percentage) {
   while (enum.hasMoreElements ()) {
      Object obj = enum.nextElement ();
      if (obj instanceof GeomShape2D) {
         GeomShape2D shape =(GeomShape2D)(obj);
         // Proceed with the algorithm...
      }
   }
}
```

Technically speaking, the only difference is the missing `else`–part in $\texttt{adjustRed}_4$. However, from an abstract viewpoint both versions implement fundamentally different algorithms. In fact, $\texttt{adjustRed}_4$ implements the algorithm,

> "modify all `GeomShape2D` objects that are found in `geom_shapes`,"

whereas $\texttt{adjustRed}_3$ implements the algorithm,

> "modify *all* items of `geom_shapes`(assuming that *all* of them are `GeomShape` objects)."

In other words, the application of `instanceof` and the down–cast are integral ingredients of the algorithm's logic in method $\texttt{adjustRed}_4$. In contrast, in $\texttt{adjustRed}_3$ they are only used to check and reconstruct certain type information, which was lost due to the type anonymity of the objects in the container `geom_shapes`.

If the test for the correct type is part of the algorithm, a quest for more static type safety certainly does not make any sense. Hence, we regard $\texttt{adjustRed}_4$ as fully statically safe. However, in the other case such a quest makes perfect sense: if the code does not compile unless all items in `geom_shapes` are of types implementing interface `GeomShape2D`, we loose nothing but gain a *much* higher degree of reliability. This is a case in which we regard an evaluation of run–time type information as statically unsafe.

We belive that the general debate on static safety suffers from a lack of accurate distinction between these two use scenarios. In fact, this distinction seems to be crucial for that discussion. Our example shows that this distinction might be rather subtle, and that the implementations of both use scenarios only differ in (seemingly) minor details, so it is not surprising that there is a lot of confusion in this debate.

In the rest of the paper, we will concentrate on the second, unsafe, use scenario.

Summary of Section 2.3

An evaluation of run–time type information that does not belong to the logic of a piece of code but is merely used to reconstruct lost type information indicates an unnecessary (and potentially dangerous) lack of static safety.

2.4 "Non–Solutions"

Clearly, an implementation of the general design pattern *adapter* [3] seems the right way of achieving loose coupling. We will come back to adapters in the very next section. In this section, we will dwell a bit on certain advanced features of Java, which also aim at loose coupling: inner classes and reflection. Besides their unquestionable merits, both of them miss both goals stated in Sections 2.2 and 2.3. It might be instructive to analyze this failure before going on with adapters.

2.4.1 Inner classes

A common idiom in Java is the usage of *inner classes* as wrappers (see Appendix B). For example, the next variant of our interface for geometric shapes, $\texttt{GeomShape2D}_4$, is based on the following wrapper class, which is named `DataWrapper`:

```
public interface DataWrapper {
   public Double getValue ();
   public void setValue (Double value);
}

public interface GeomShape2D4 {
   public DataWrapper getRedWrapper ();
   public DataWrapper getGreenWrapper ();
   public DataWrapper getBlueWrapper ();

   // Further general methods for arbitrary two-dimensional shapes,
   // e.g. for shifting or rotating a shape object
}
```

The idea is this: `getRedWrapper` returns a `DataWrapper` whose methods read and overwrite the red color of the corresponding shape object (`getGreenWrapper` and `getBlueWrapper` analogously). The following implementation of the interface `GeomShape2D4` demonstrates this technique:

```
public class GeomShape2DWithColorWrappers
implements GeomShape2D4 {
   Double red;
   Double green;
   Double blue;
   // Color wrappers:
   public class RedWrapper implements DataWrapper {
      public Double getValue () {
         return GeomShape2DWithColorWrappers.this.red;
      }
      public void setValue (Double red) {
         GeomShape2DWithColorWrappers.this.red = red;
      }
   }
   public DataWrapper getRedWrapper () {
      return new RedWrapper ();
   }

   // Analogous inner classes Green/BlueWrapper and methods getGreen/BlueWrapper
   // Further methods of GeomShape2D4
}
```

An implementation of `adjustRed` in which inner classes encapsulate the method access:

```
public static void adjustRed5(Enumeration enum, double percentage) {
   while (enum.hasMoreElements ()) {
      Object obj = enum.nextElement ();
      if (obj instanceof GeomShape2D4) {
         GeomShape2D4 shape =(GeomShape2D4)(obj);
         DataWrapper red_wrapper = shape.getRedWrapper ();
         DataWrapper green_wrapper = shape.getGreenWrapper ();
         DataWrapper blue_wrapper = shape.getBlueWrapper ();
         double red = red_wrapper.getValue ().doubleValue ();
         if (red > 1-percentage/100) {
            red_wrapper.setValue (new Double (1));
            green_wrapper.setValue (new Double (0));
            blue_wrapper.setValue (new Double (0));
         } else {
            // etc.
         }
      } else throw MyFavoriteException ();
   }
}
```

It might be obvious from this example that inner classes do not really solve the problem we addressed in Section 2.2.[4] In fact, the strong coupling between the client `adjustRed`$_5$ and the underlying geometric shape type is simply shifted from the original methods (`get/setRed` etc.) to the new methods (`getRed/ Green/ BlueWrapper`). In view of Section 2.2, it makes no difference whether the identifier `getRed` or the identifier `getRedWrapper` is "hard–coded" in `adjustRed`$_5$.

2.4.2 Reflection

Run–time type information can be used to access all methods of a class without knowing in advance which methods this class offers and which signatures they have. In Java, reflection is implemented by the *reflection API*[5]. This package cannot handle primitive types, which is one of the reasons why the design of the case study from Section 2.1 relies on class `Double` instead of primitive type `double`.

A variant of method `adjustRed` based on reflection (sketched) can be seen in figure 2.5.

Clearly, static safety is completely missed. On the other hand, the names

[4] At first glance, there seems to be an elegant workaround: the methods `getRed/ Green/ BlueWrapper` are not called inside `adjustRed`$_5$. Instead, `adjustRed`$_5$ has three enumerations arguments, which refer to sequences of `DataWrapper` objects for all three colors. More specifically, for every item in the original sequence `geom_shapes` there is a `DataWrapper` object in each of the new sequences, which refers to the original item. However, static safety is still missing, and a really loose coupling is not achieved either, because `adjustRed`$_5$ still depends on the existence of inner classes implementing `DataWrapper`. In other words, `adjustRed`$_5$ would not depend on the exact signatures of `getRed/ Green/ BlueWrapper` anymore but still on the

Figure 2.5 A variant of method `adjustRed` based on reflection (sketched).

```
public static void adjustRed6(Enumeration enum,
   String name_of_red_get_method, String name_of_red_set_method,
   String name_of_green_get_method, String name_of_green_set_method,
   String name_of_blue_get_method, String name_of_blue_set_method,
   double percentage) {
   String argument_types_of_red_get = new Class [0];
   String argument_types_of_red_set = new Class [1];
   String argument_values_of_red_get = new Object [0];
   String argument_values_of_red_set = new Object [1];
   try {
      argument_types_of_red_set[0] = Class .forName (”java.lang.Double ”);
   }
   catch (Exception e) {
      // Do some reasonable exception handling
   }
   // Analogously green and blue

   while (enum.hasMoreElements ()) {
      Object obj = enum.nextElement ();
      java.lang.reflect.Method get_red;
      java.lang.reflect.Method set_red;
      try {
         get_red = obj.getClass ().getMethod (name_of_red_get_method,argument_types_of_red_get);
         set_red = obj.getClass ().getMethod (name_of_red_set_method, argument_types_of_red_set);
      }
      catch (Exception e) {
         // Do some reasonable exception handling
      }
      // Analogously green and blue

      Double red_obj = get_red.invoke (obj, argument_values_of_red_get);
      double red = red_obj.doubleValue ();
      if (red > 1-percentage/100) {
         set_red.invoke (obj, new Double (1));
         set_green.invoke (obj, new Double (0));
         set_blue.invoke (obj, new Double (0));
      } else {
         // etc.
      }
   }
}
```

and signatures of the color–accessing methods are not hard–wired in the code of adjustRed$_6$. However, we note that adjustRed$_6$ is not really independent of these methods, because the number of arguments of these methods are still hard–wired.[6]

> **Summary of Section 2.4**
> Even advanced features such as inner classes and reflections, which are specifically intended to implement loose coupling, are not sufficient to achieve the goals in Sections 2.2 and 2.3 simultaneously.

2.5 Adaptation

The adapter pattern [3] can be used to decouple a class from its clients. In our case study, this means that a subroutine such as adjustRed is not based on the comprehensive interface GeomShape2D, which captures various aspects of geometric shapes, but on a small interface, which only captures the aspects relevant for adjustRed, for example, leaned on the signature of GeomShape2D (see figure 2.6 for the corresponding UML-diagram):

```
public interface RGBHandler {
    public double getRed (Object obj);
    public double getGreen (Object obj);
    public double getBlue (Object obj);
    public void setRed (Object obj, Double r);
    public void setGreen (Object obj, Double g);
    public void setBlue (Object obj, Double b);
}
```

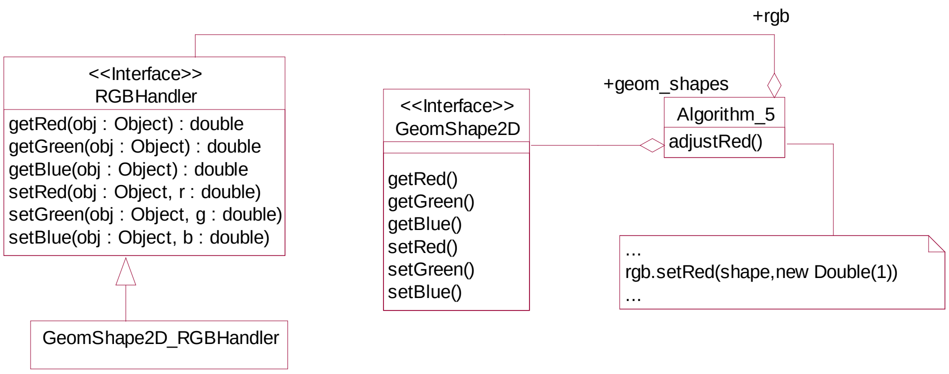
A variant of adjustRed based on RGBHandler could look like this (sketched):

existence of these three methods (having whatever signatures).

[5] See Appendix C or java.lang.reflect.*.

[6] In principle, Java's reflection mechanism is powerful enough even to render the number of arguments variable. However, then the problem remains what adjustRed$_6$ should do in case of, say, additional arguments, that is, what values they should be assigned by adjustRed$_6$.

Figure 2.6 UML-diagram for rgb handlers




```
public static void adjustRed_7(Enumeration enum,
        RGBHandler rgb, double percentage) {
    while (enum.hasMoreElements ()) {
        Object obj = enum.nextElement ();
        double red = rgb.getRed (obj).doubleValue ();
        if (red > 1-percentage/100) {
            rgb.setRed (obj, new Double (1));
            rgb.setGreen (obj, new Double (0));
            rgb.setBlue (obj, new Double (0));
        } else {
            // etc.
        }
    }
}
```

Notice that down–casts are not avoided but moved from the algorithm to the adapting class. To apply `adjustRed` to a class like `GeomShape2D`, a class `GeomShape2D_RGBHandler`, say, is defined, which implements `RGBHandler` and “knows” all relevant details of `GeomShape2D`:

```
public class GeomShape2D_RGBHandler implements RGBHandler {
   public Double getRed (Object obj) {
      if (obj instanceof GeomShape2D) {
         return ((GeomShape2D)obj).getRed ();
      else throw MyFavoriteException ();
   }

   public void setRed (Object obj, Double r) {
      if (obj instanceof GeomShape2D) {
         ((GeomShape2D)obj).setRed (r);
      else throw MyFavoriteException ();
   }
   // Green and blue analogously
}
```

Now we can apply $\texttt{adjustRed}_7$ to a sequence of `GeomShape2D` objects:

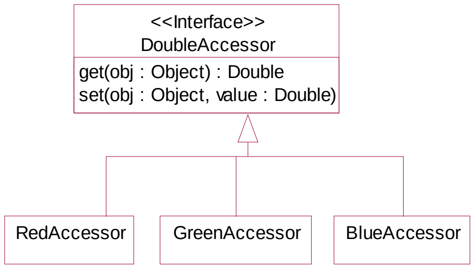
```
GeomShape2D_RGBHandler rgb = new GeomShape2D_RGBHandler;
GeomAlgorithms.adjustRed7(geom_shapes.elements (), rgb, 10.0);
```

In general, the individual attributes of a class are not as strongly coupled as the RGB color values of a geometric shape. Hence, in general it might be preferable to provide one separate handler object for each attribute. Such a one–to–one correspondence between attributes and handlers would exactly implement the *data–accessor* concept as introduced in [4] or [5] and discussed in [7]. Note however, that although adapter and data accessor are similar on the implementation side, they are completely different in their intent (see [5] for a detailed discussion): the intent of data accessors is to encapsulate the access to attributes of classes in small, light–weight classes. This allows a common, uniform interface for all attributes of all classes, which means that classes and attributes are easily exchangeable in an attribute–accessing client such as `adjustRed`.[7] To emphasize that the following data–accessor interface only applies to attributes of type `Double`, we will call the interface `DoubleAccessor` (see figure 2.7 for the UML-diagram):

```
public interface DoubleAccessor {
   public Double get (Object obj);
   public void set (Object obj, Double value);
}
```

[7]Note that the *JavaBeans* convention for method signatures [6] does *not* provide a uniform interface to attributes in the strong sense as used in this paper. In the *JavaBeans* approach, the signatures of a pair of `get/set` methods depend on the name of the attribute in a disciplined manner. In contrast, the signatures of the `get/set` methods of a data accessor do not at all depend on the name and type of the attribute. This difference results from different goals: the *JavaBeans* approach allows an easy *access* of each attribute given the name of the attribute, whereas we need a convention that renders the name of the attribute completely anonymous to allow an easy *exchange*.

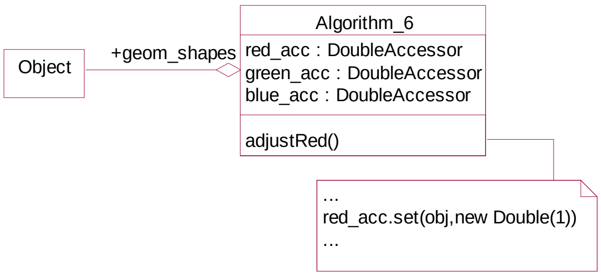
Figure 2.7 UML-diagram for double accessor



And the corresponding implementation of `adjustRed` (see figure 2.8 for a UML-diagram):

```
public static void adjustRed8(Enumeration enum, DoubleAccessor red_acc,
   DoubleAccessor green_acc, DoubleAccessor blue_acc, double percentage) {
   while (enum.hasMoreElements ()) {
      Object obj = enum.nextElement ();
      double red = red_acc.get (obj).doubleValue ();
      if (red > 1-percentage/100) {
         red_acc.set (obj, Double (1));
         green_acc.set (obj, Double (0));
         blue_acc.set (obj, Double (0));
      } else {
         // etc.
      }
   }
}
```

Figure 2.8 UML-diagram for an algorithm that uses double accessors



For ease of exposition, we will use the variant `adjustRed`$_8$ in the rest of the paper.

2.6 Data Accessors and Down–Casts

Unfortunately, using data accessors in Java results in many down–casts due to the fact that the object type is left open in the interface `DoubleAccessor` (it is **Object**). We will illustrate this problem by sketching the implementation of an appropriate data–accessor class for the basic scenario from Section 2.1 and for the variations from Section 2.2.

The crucial impression to be taken from the examples is this: *each scenario will require a down–cast of the kind that indicates an unnecessary lack of static safety*. For ease of exposition, we omit all exception handling, and we only give the accessors for the red color value.

2.6.1 Data Accessor for Straightforward Implementation

Here is a data–accessor class for the red color value based on interface `GeomShape2D` from Section 2.1, page 5:

```
public class RedAccessor1 implements DoubleAccessor {
   public Double get (Object obj) {
      return ((GeomShape2D)obj).getRed ();
   }
   public Double set (Object obj, Double value) {
      ((GeomShape2D)obj).setRed (value);
   }
}
```

2.6.2 First variation: combined attributes

For `GeomShape2D`$_2$(*i.e.* first variation in Section 2.2, page 2.2.1):

```
public class RedAccessor2
implements DoubleAccessor {
   public Double get (Object obj) {
      GeomShape2D2 shape =(GeomShape2D2) obj;
      return shape.getColor ().getRedPart ();
   }
   public Double set (Object obj, Double value) {
      GeomShape2D2 shape =(GeomShape2D2) obj;
      shape.getColor ().setRedPart (value);
   }
}
```

2.6.3 Second Variation: Dictionaries

The data–accessor class for the second variation from Section 2.2, page 2.2.2 is somewhat different. Since the color values are stored outside the shape class, they must be handed over to the data accessors in a way that does not affect the code of $\texttt{adjustRed}_8$. In other words, the data accessors must receive the color values before $\texttt{adjustRed}_8$ is called (*e.g.* as an argument to the constructor). Interestingly, this particular data–accessor class is not bound to a specific implementation of geometric shapes, not even to a specific attribute, because the shape object is merely handed over anonymously to the dictionary of attribute values. In fact, `DoubleDictAccessor` only depends on the attribute type `Double`, which enforces another down–cast:

```
public class DoubleDictAccessor
implements DoubleAccessor {
   Dictionary dictionary;
   public DoubleDictAccessor (Dictionary dictionary) {
      this.dictionary = dictionary;
   }
   public Double get (Object obj) {
      Object item = dictionary.get (obj);
      return (Double )item;
   }
   public void set (Object obj, Double value) {
      dictionary.put (obj, value);
   }
}
```

2.6.4 Third variation: different encoding

Next we consider a data–accessor class for $\texttt{GeomShape2D}_3$ (third variation in Section 2.2, page 11). The following implementation, `RedHSBAccessor`, is based on three further data accessors, which access the hue, saturation, and brightness value of the geometric shape object, respectively. Hence, this class, which performs non–trivial algorithmic tasks,[8] is also decoupled from all "volatile" details of the underlying shape class and might itself be better maintainable and reusable.

[8]For ease of exposition, the conversion algorithms RGB$\leftrightarrow$HSB are integrated in the data–accessor class. In a more realistic design they would be implemented as methods of a separate class.

```
public class RedAccessor3
implements DoubleAccessor {
  DoubleAccessor hue_acc;
  DoubleAccessor saturation_acc;
  DoubleAccessor brightness_acc;
  public RedAccessor3(DoubleAccessor hue_acc,
    DoubleAccessor saturation_acc, DoubleAccessor brightness_acc) {
    this.hue_acc = hue_acc;
    this.saturation_acc = saturation_acc;
    this.brightness_acc = brightness_acc;
  }
  public Double get (Object obj) {
    Double hue = hue_acc.get (obj);
    Double saturation = saturation_acc.get (obj);
    Double brightness = brightness_acc.get (obj);
    Double red = /* Red part of result HSB → RGB */
    return red;
  }
  public void set (Object obj, Double value) {
    Double hue = hue_acc.get (obj);
    Double saturation = saturation_acc.get (obj);
    Double brightness = brightness_acc.get (obj);
    Double green;
    Double blue;
    /* Compute green and blue from hue, saturation, and brightness */
    /* Compute the new values of hue, saturation,
      and brightness from the RGB triple (value,green,blue) */
    hue_acc.set (obj, hue);
    saturation_acc.set (obj, saturation);
    brightness_acc.set (obj, brightness);
  }
}
```

2.6.5 Inner classes

For completeness, we will also show how to bring interface `GeomShape2D4` from Section 2.4 into the game:

```
public class RedAccessor4
implements DoubleAccessor {
   public Double get (Object obj) {
      GeomShape2D4 shape =(GeomShape2D4)obj;
      return shape.getRedWrapper ().getValue ();
   }
   public void set (Object obj, Double value) {
      GeomShape2D4 shape =(GeomShape2D4)obj;
      shape.getRedWrapper ().setValue (value);
   }
}
```

2.6.6 Reflection

To conclude this section, it might be instructive to see how a data accessor may be based on reflection instead of mere down–casts. Of course, this does not change the overall conclusion that there remains a lack of static safety.

```
public class ReflectionAccessor
implements DoubleAccessor {
   String name_of_get_method, name_of_set_method;
   Class [] argument_types_of_get, Class [] argument_types_of_set;
   Object [] argument_values_of_get, Object [] argument_values_of_set;
   ReflectionAccessor (String name_of_get_method,
   String name_of_set_method) {
      this.name_of_get_method = name_of_get_method;
      this.name_of_set_method = name_of_set_method;
      argument_types_of_get = new Class [0];
      argument_types_of_set = new Class [1];
      argument_values_of_get = new Object [0];
      argument_values_of_set = new Object [1];
      try {
         argument_types_of_set[0] = Class .forName (”java.lang.Double ”);
      }
      catch (Exception e) {
         // Do some reasonable exception handling
      }
   }
}
   public Double get (Object obj) {
      GeomShape2D3 shape =(GeomShape2D3 obj;
      try {
         Method get = shape.getClass ().getMethod (name_of_get_method, argument_types_of_get);
         Object return_obj = get.invoke (shape, argument_values_of_get);
         return (Double )return_obj;
      } catch (Exception e) {
         // Do some reasonable exception handling
      }
   }
   public void set (Object obj, Double value) {
      GeomShape2D3 shape =(GeomShape2D3 obj;
      argument_values_of_set[0] = value;
      try {
         Method set = shape.getClass ().getMethod (name_of_set_method, argument_types_of_set);
         set.invoke (shape, argument_values_of_set);
      } catch (Exception e) {
         // Do some reasonable exception handling
      }
   }
}
```

Summary of Section 2.6

In pure Java, a true open–world design results in a serious lack of static safety (indicated by down–casts or other ways of evaluating run–time type information). Adapters may *encapsulate* but not *remove* this gap.

2.7 Making the Attribute Type Generic

So far, we did not vary the types of the attributes to work out the crucial point more clearly: even if the type of an attribute is fixed, an open–world design of attribute access results in a serious type–safety problem. In this section, we will require for true "open–worldness" that the attribute type is also left variable. It is not surprising that static safety will be seriously affected. However, the extent to which this will happen might be surprising: we will have to introduce an additional, auxiliary interface (named `Traits` below), and the attribute type and the traits type have to fit together exactly.

To make our algorithm applicable to various attribute types, we have to replace `Double` by a more general type; `Object` is the prime candidate for that:

```
public interface ObjectAccessor {
   public Object get (Object obj);
   public void set (Object obj, Object value);
}
```

Here is an appropriate implementation of `ObjectAccessor` for the red color value of `GeomShape2D`:

```
public class ObjectAccessorRedGeomShape2D
implements ObjectAccessor {
   public Object get (Object obj) {
      GeomShape2D shape =(GeomShape2D)obj;
      return shape.getRed ();
   }
   public void set (Object obj, Object value) {
      GeomShape2D shape =(GeomShape2D)obj;
      Double val_obj =(Double )value;
      shape.setRed (val_obj);
   }
}
```

So far, nothing changed. The new problem is due to the fact that our algorithm applies certain operations to objects of the attribute type: the four basic numerical operations and a comparison operation. In view of Section 2.2, we cannot assume that all potential attribute types provide a common signature for all of these operations. Hence, to write down these operations without knowing the concrete attribute type, we collect these operations in an additional object `traits`.[9] In the following, a traits object is an instance of a class that implements the following interface:

```
public interface Traits {
   public Object generate (double x);
   public Object plus (Object val1, Object val2);
   public Object minus (Object val1, Object val2);
   public Object mult (Object val1, Object val2);
   public Object div (Object val1, Object val2);
   public boolean isGreaterThan (Object val1, Object val2);
}
```

The first method returns a reference to an object of the anonymous attribute type, and the value of this object shall represent the value of `x`. We will only use this method to generate objects representing `0`, `1`, and `100`, respectively. The requirement that these three values are feasible might not cause any problem for any class type that represents real numbers.

The other methods assume that their arguments are of the attribute type. Methods #2–6 are also required to return references to objects of the attribute type. These methods perform the basic arithmetical operations on the attribute type. Finally, the last method returns `true` if and only if the first argument is to be regarded as greater than the second argument.

Now we are able to formulate a version of `adjustRed` in which the attribute type is left open. A brief sketch:

[9] The name is borrowed from the analogous *traits* idiom in C++, which is heavily used in the standard library.

```
public void adjustRed9(Enumeration enum, ObjectAccessor red_acc,
ObjectAccessor green_acc, ObjectAccessor blue_acc,
Object percentage, Traits traits) {
   Object zero = traits.generate (0);
   Object one = traits.generate (1);
   Object hundred = traits.generate (100);
   Object fraction = traits.div (percentage, hundred);
   Object threshold = traits.minus (one, fraction);
   while (enum.hasMoreElements ()) {
      Object obj = enum.nextElement ();
      Object red = red_acc.get (obj);
      if (traits.isGreaterThan (red, threshold)) {
         obj.setRed (one);
         obj.setGreen (zero);
         obj.setBlue (zero);
      } else {
         // etc.
      }
   }
}
```

Here is a concrete example of such a traits class, which would perfectly collaborate with `ObjectAccessorRed/Green/BlueGeomShape2D` via class `Double`:

```
public class DoubleTraits
implements Traits {
   public Object generate (double x) {
      return new Double (x);
   }
   public Object plus (Object val1, Object val2) {
      double d1 =((Double )val1).doubleValue ();
      double d2 =((Double )val2).doubleValue ();
      return new Double (d1+d2);
   }
   // Analogously: minus, mult, div, isGreaterThan
}
```

Such a design requires even more care, because it must be additionally guaranteed *by the software developer* that the attribute type and the traits type fit correctly together. Clearly, this kind of unsafe collaborations increases the potential for pitfalls dramatically and is much harder to debug.

Summary of Section 2.7

A generic attribute type requires an unsafe collaboration between the attribute type itself and additional, auxiliary types.

2.8 Parametric Polymorphism: GJ

Of course, the best way to avoid trouble with down–casts is to avoid down–casts at all. However, it seems that in plain Java, down–casts cannot be avoided, unless all data structures are "hard–wired" in the algorithms. Clearly, this would destroy all hope even for a rudimentary form of open–world design. As we saw, this conflict does not seem to be resolvable in plain Java. In this section, we will show that it is resolvable in generic language extensions such as GJ [2]. The essential feature missing in Java is *parametric polymorphism*(see Appendix D). We will start right from Section 2.7, because the genericity of the attribute type will come as a by–product.

To begin with, we replace the interface `ObjectAccessor` from Section 2.7 by the interface `GenericAccessor`, which is equally general, but will not enforce any down–casts[10] in the classes implementing this interface:

```
public interface
GenericAccessor<ObjectType,ValueType> {
   public ValueType get (ObjectType obj);
   public void set (ObjectType obj, ValueType value);
}
```

The formal type argument `ObjectType` stands for the class to which the attribute is associated (*i.e.* `GeomShape2D` in our running example), whereas `ValueType` stands for the attribute type. The goal is to use `GenericAccessor` to eliminate the need for `Object` in the definition of `adjustRed`. Since `java.lang.Enumeration` also works on `Object`, we have to replace `Enumeration` by a variant that is parameterized by the item type of the underlying container:

```
public interface
GenericEnumeration<ObjectType> {
   public boolean hasMoreElements ();
   public ObjectType nextElement ();
}
```

Now, our `Vector` object `geom_shapes` may be replaced by a vector that is specific to the interface `GeomShape2D`:

[10]According to [2], generic type parameters are internally realized by down–casts in GJ. However, this is an implementation detail of the compiler and does not affect the static safety offered to the developer.

```
Vector <GeomShape2D> geom_shapes = new Vector <GeomShape2D>();
```

The traits interface introduced in Section 2.7 is also replaced by a generic one:

```
public interface GenericTraits <ValueType> {
   public static ValueType generate (double x);
   public static ValueType plus (ValueType val1, ValueType val2);
   public static ValueType minus (ValueType val1, ValueType val2);
   public static ValueType mult (ValueType val1, ValueType val2);
   public static ValueType div (ValueType val1, ValueType val2);
   public static boolean isGreaterThan (ValueType val1, ValueType val2);
}
```

Now we are in a position to implement a truly generic version of `adjustRed`, which we call $\texttt{adjustRed}_{10}$. This algorithm will be a method of a variant of class `GeomAlgorithms` from Section 2, which is parameterized by `ObjectType` and `ValueType`:

```
public class
GeomAlgorithms2<ObjectType, ValueType> {
   public static void adjustRed10 (GenericEnumeration<ObjectType> enum,
      GenericAccessor<ObjectType,ValueType> red_acc,
      GenericAccessor<ObjectType,ValueType> green_acc,
      GenericAccessor<ObjectType,ValueType> blue_acc,
      ValueType percentage,
      GenericTraits<ValueType> traits) {
      ValueType zero      = traits.generate (0);
      ValueType one       = traits.generate (1);
      ValueType hundred   = traits.generate (100);
      ValueType fraction  = traits.div (percentage, hundred);
      ValueType threshold = traits.minus (one, fraction);
      while (enum.hasMoreElements ()) {
         ObjectType shape = enum.nextElement ();
         ValueType red = red_acc.get (shape);
         if (traits.isGreaterThan (red, threshold)) {
            shape.setRed (one);
            shape.setGreen (zero);
            shape.setBlue (zero);
         } else {
            // etc.
         }
      }
   }
}
```

The following variant is a concrete example of how the interface `Generi-`

`cAccessor` may be implemented in order to access a concrete attribute of a concrete class/interface such as `GeomShape2D`.

```
public class AccessorRedGeomShape2D
implements GenericAccessor<GeomShape2D,Double > {
   public Double get (GeomShape2D shape) {
      return shape.getRed ();
   }
   public void set (GeomShape2D shape, Double value) {
      shape.setRed (value);
   }
}
```

To realize the last argument of $\texttt{adjustRed}_{10}$, we let a variant of `Double-Traits`(see Section 2.7) implement `GenericTraits<Double>`:

```
public class DoubleTraits2
implements GenericTraits<Double > {
   public Double generate (double x) {
      return new Double (x);
   }
   public Double plus (Double val1, Double val2) {
      double d1 = val1.doubleValue ();
      double d2 = val2.doubleValue ();
      return new Double (d1+d2);
   }
   // Analogously minus, mult, div, and isGreaterThan
}
```

Finally, here is the call to the generic version of `adjustRed`:

```
AccessorRedGeomShape2D red_acc       = new AccessorRedGeomShape2D ();
AccessorGreenGeomShape2D green_acc   = new AccessorGreenGeomShape2D ();
AccessorBlueGeomShape2D blue_acc     = new AccessorBlueGeomShape2D ();
```

```
DoubleTraits2 traits = new DoubleTraits2();

GeomAlgorithms2 adjustRed10 (geom_shapes.elements (),
   red_acc, green_acc, blue_acc, 10.0, traits);
```

These code snippets might give an idea how parametric polymorphism may be applied in the second use scenario of Section 2.2. The result is as flexible as possible, absolutely safe, and even simpler than any of the previous designs in plain Java.

3 Conclusion

Figure 3.1 variations of data accessor implementations

	object type fixed value type fixed	object type variable value type fixed	object type variable value type variable
pure Java	ok	no static type safety	no static type safety use of traits
Java + GJ	ok	ok	use of traits
C++	ok	ok	ok

In view of maintenance and reuse, it is desirable to render the coupling between collaborating classes as loose as possible, but nonetheless (statically) safe. We have seen that this is a serious problem even for the access to a single attribute via a pair of `get/set` methods (even if the attribute type is fixed). The language features of Java cannot resolve the contradiction between loose coupling and safety (see figure 3.1 for a summary of the discussion). With a generic language extension like GJ or by using `C++`, it is possible to enable static type safety. However, only in `C++`, the traits mechanism can be avoided by using operator overloading.

We have identified a key feature, which can help to overcome this conflict: parametric polymorphism (genericity). As we have seen, this feature is helpful beyond its original intention (namely to implement parameterized algorithms and data structures). In our opinion, this is a strong argument that genericity should be treated as a first–class design feature rather than a mere "implementation trick" for type–independent algorithms and data structures.

An in–depth discussion of genericity as a design feature is beyond the scope of this paper. A single paper of this length cannot give more than a base for such a discussion.

Acknowledgments: we would like to thank Christoph Burnikel for valuable comments.

Bibliography

[1] Ole Agesen, Stephen Freund, and John C. Mitchell:
Adding Type Parameterization to Java
Proceedings of the 12th ACM Symposium on Object–Oriented Programming, Systems, Languages, and Applications (OOPSLA '97), 49–65

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler:
Making the Future Safe for the Past: Adding Genericity to the Java Programming Language
Proceedings of the 13th ACM Symposium on Object–Oriented Programming, Systems, Languages, and Applications (OOPSLA '98),

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides:
Design Patterns – Elements of Reusable Object–Oriented Software
Addison–Wesley, 1995

[4] Dietmar Kühl and Karsten Weihe:
Data Access Templates
C++ Report 9/7 (1997), 15–21

[5] Marco Nissen:
Design Pattern Data Accessor
Proceedings of the 4th European Conference on Pattern Languages of Programming (EuroPLoP '99)

[6] Peter Wayner:
Java Beans for Real Programmers
AP Professional, 1998

[7] Karsten Weihe:
Reuse of Algorithms: Still a Challenge to Object–Oriented Programming
Proceedings of the 12th ACM Symposium on Object–Oriented Programming, Systems, Languages, and Applications (OOPSLA '97), 34–48

Appendix

Appendix A: Java Enumerations

Every container class in the Java utility library provides a method `elements ()`, which returns a reference to the interface `java.util.Enumeration`. This is the standard interface for iterator classes in Java. For example, a traversal of a vector `v` may be implemented like this:

```
Vector v = new Vector ();
/* ... */

Enumeration enum = v.elements ();
while (enum.hasMoreElements ()) {
   Object obj = enum.nextElement ();
   // Do something reasonable with obj
}
```

Clearly, for every self–defined container class, one can also implement a specific `Enumeration` class. Hence, an algorithm that accesses containers exclusively through interface `Enumeration` is completely independent of the choice of the container class. This greatly improves maintainability and reusability.

Appendix B: Inner Classes

A *nested class* is a class that is defined inside another class. A nested, non–static class is commonly called an *inner class*. Here is a simple, illustrative example:

```
public class OuterClass {
   int n;
   public class InnerClass {
      public int get () {
         return n;
      }
   }
}
```

Note that `InnerClass` is allowed to access non–static — even private — members of the class in which it is embedded (as opposed to nested classes in C++). Inner classes are useful for many purposes. For example, they can be used to circumvent the restriction that no class may inherit from more than one class. Inner classes are also useful for lean implementations of adapters. The interested reader is referred to the overwhelming literature on the Java language.

The aspect in which we are specifically interested in view of our case study is the selection of class attributes. For sake of exposition, consider a simple business model, which is based on three classes, `Employee`, `Customer`, and `Freelance`. Among other attributes, we keep the employees' salaries, the customers' credits and debits, and the freelances' contractual payments. Some accounting algorithms (*e.g.* calculation of interest) might be generic in the sense that they are useful for each of these attributes. At first glance, it suffices to derive employees, customers, and freelances from a common base class or interface `PersonWithMoneyAttribute`, say, which has a generic money attribute, and to implement every accounting algorithm on top of `PersonWithMoneyAttribute` using this money attribute. However, customers have *two* money attributes, so this idea simply fails. Even if customers only had one money attribute: a design that regards salaries, credits or debits, and contractual payments as conceptually identical might not be sound and thus should be avoided.

Inner classes offer a solution to this dilemma: every person class implements an inner class for every money attribute. All of these classes implement a common interface `MoneyHandler`:

```
public interface MoneyHandler {
   MoneyType get ();
   void set (MoneyType amount);
}
```

For example, class `Customer` would define one inner class for credits and one for debits:

```
class Customer {
   public class CreditHandler
   implements MoneyHandler {
      MoneyType get () {
         /* ... */
   }
      void set (MoneyType amount) {
         /* ... */
      }
   }
   public CreditHandler getCreditHandler () {
      return new CreditHandler ();
   }
   public class DebitHandler
   implements MoneyHandler {
      MoneyType get () {
         /* ... */
      }
      void set (MoneyType amount) {
         /* ... */
      }
   }
   public DebitHandler getDebitHandler () {
      return new DebitHandler ();
   }
   // Further stuff
}
```

An accounting algorithm may then be implemented on top of `MoneyHandler`:

```
MoneyType myAccountingAlgorithm (MoneyHandler mh) {
   /* ... */
}
```

Appendix C: Reflection

The package `java.lang.reflect` provides a means of analyzing objects of unknown classes at run time. For example, this includes means of retrieving the name of an object's class, the names and types of its data members, and the names and argument lists of its methods. In this paper, we are particularly interested in another powerful feature: invoking a method whose signature is only known at run time. For example, the following method invokes a method of `obj` with one argument. The name of the method is given in the string `name_of_method`, the type of the only argument is `argument_type`, and `value` is to be used as the value of this argument when the method is invoked. The classes `Class` and `Method` are defined in `java.lang.Class` and `java.lang.reflect.Method`, respectively. As the class names indicate, an object of class `Class` (resp. `Method`) contains general information about a particular class (method of a class). Method `forName` of `Class` turns the (fully qualified) name of a class into an object for that class.

```
public void invoke_method (Object obj, String name_of_method,
                           String argument_type, Object value) {
  Class obj_class_id = obj.getClass ();
  Class val_class_id = value.getClass ();
  String val_class_name = val_class_id.getName ();
  Class [] argument_types = new Class [1];
  Object [] argument_values = new Object [1];
  try {
    argument_types[0] = Class .forName (argument_type);
    argument_values[0] = value;
    Method method = obj_class_id.getMethod
      (name_of_method, argument_types);
    method.invoke (obj, argument_values);
  }
  catch (Exception e) {
    // Do some reasonable exception handling
  }
}
```

Appendix D: Parametric Polymorphism (GJ)

GJ's [2] parametric polymorphism is very similar to *templates* in C++, *generics* in Ada, parametric polymorphism in functional languages, and various other generic extensions of Java [1]. This means that the concrete types on which a class definition is based may be left open. Such an "incomplete" class is called a *parameterized class*. In the definition of such a parameterized class, these types are represented by *formal type arguments*. Roughly speaking, a formal type argument is a placeholder for the actual type. The actual type must be specified when an object of the parameterized class is instantiated.

To give a simple example, the following stack class is parameterized by the type `T` of its items. In GJ style:

```
public class Stack <T> {
   public Stack () { /* ... */ }
   public void push (T t) { /* ... */ }
   public T top () { /* ... */ }
   public void pop () { /* ... */ }
   public int size () { /* ... */ }
}
```

GJ does not allow that `T` is a primitive type such as `int` or `double`. In fact, `T` must be a class or interface (this is another reason why the case study from Section 2 uses class `Double` instead of primitive type `double`). The following code snippet demonstrates how a stack of `Integer` may be created and used:

```
Stack<Integer > S1 = new Stack<Integer >();
S1.push (new Integer (1));
// Should print '1':
System.out.println (S1.top ().intValue ());
S1.push (new Double (2));
// Compiler error: wrong type
```

Note that the compiler checks whether or not an object of the parameterized stack class is correct according to the concrete item type `T`: if we try to push a value of a wrong item type onto a stack, the compiler issues an error message. This is in great contrast to a generic stack class in plain Java, which must be based on `comp.lang.Object`:

```
public class Stack {
   public Stack () {
      /* ... */
   }
   public void push (Object obj) {
      /* ... */
   }
   public Object top () {
      /* ... */
   }
   public void pop () {
      /* ... */
   }
   public int size () {
      /* ... */
   }
}

/* ... */

Stack S; // Intended to be a stack of Integer

/* ... */

S.push (new Double (5));
// Oops: no compiler error; S silently becomes inconsistent!
```

This is an example of the first, unsafe, use scenario in Section 2.3. This observation can be generalized: the cases for which genericity was originally introduced are a special case of the second use scenario.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Birgit Hofmann
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron
MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid *E*-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-2-002	F. Jacquemard, C. Meyer, C. Weidenbach	Unification in Extensions of Shallow Equational Theories
MPI-I-98-1-031	G.W. Klau, P. Mutzel	Optimal Compaction of Orthogonal Grid Drawings
MPI-I-98-1-030	H. Brönniman, L. Kettner, S. Schirra, R. Veltkamp	Applications of the Generic Programming Paradigm in the Design of CGAL
MPI-I-98-1-029	P. Mutzel, R. Weiskircher	Optimizing Over All Combinatorial Embeddings of a Planar Graph

MPI-I-98-1-028	A. Crauser, K. Mehlhorn, E. Althaus, K. Brengel, T. Buchheit, J. Keller, H. Krone, O. Lambert, R. Schulte, S. Thiel, M. Westphal, R. Wirth	On the performance of LEDA-SM
MPI-I-98-1-027	C. Burnikel	Delaunay Graphs by Divide and Conquer
MPI-I-98-1-026	K. Jansen, L. Porkolab	Improved Approximation Schemes for Scheduling Unrelated Parallel Machines
MPI-I-98-1-025	K. Jansen, L. Porkolab	Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks
MPI-I-98-1-024	S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, M. Vingron	q-gram Based Database Searching Using a Suffix Array (QUASAR)
MPI-I-98-1-023	C. Burnikel	Rational Points on Circles
MPI-I-98-1-022	C. Burnikel, J. Ziegler	Fast Recursive Division
MPI-I-98-1-021	S. Albers, G. Schmidt	Scheduling with Unexpected Machine Breakdowns
MPI-I-98-1-020	C. Rüb	On Wallace's Method for the Generation of Normal Variates
MPI-I-98-1-019		2nd Workshop on Algorithm Engineering WAE '98 - Proceedings
MPI-I-98-1-018	D. Dubhashi, D. Ranjan	On Positive Influence and Negative Dependence
MPI-I-98-1-017	A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, E. Ramos	Randomized External-Memory Algorithms for Some Geometric Problems
MPI-I-98-1-016	P. Krysta, K. Loryś	New Approximation Algorithms for the Achromatic Number
MPI-I-98-1-015	M.R. Henzinger, S. Leonardi	Scheduling Multicasts on Unit-Capacity Trees and Meshes
MPI-I-98-1-014	U. Meyer, J.F. Sibeyn	Time-Independent Gossiping on Full-Port Tori
MPI-I-98-1-013	G.W. Klau, P. Mutzel	Quasi-Orthogonal Drawing of Planar Graphs
MPI-I-98-1-012	S. Mahajan, E.A. Ramos, K.V. Subrahmanyam	Solving some discrepancy problems in NC*
MPI-I-98-1-011	G.N. Frederickson, R. Solis-Oba	Robustness analysis in combinatorial optimization
MPI-I-98-1-010	R. Solis-Oba	2-Approximation algorithm for finding a spanning tree with maximum number of leaves
MPI-I-98-1-009	D. Frigioni, A. Marchetti-Spaccamela, U. Nanni	Fully dynamic shortest paths and negative cycle detection on diagraphs with Arbitrary Arc Weights
MPI-I-98-1-008	M. Jünger, S. Leipert, P. Mutzel	A Note on Computing a Maximal Planar Subgraph using PQ-Trees
MPI-I-98-1-007	A. Fabri, G. Giezeman, L. Kettner, S. Schirra, S. Schönherr	On the Design of CGAL, the Computational Geometry Algorithms Library
MPI-I-98-1-006	K. Jansen	A new characterization for parity graphs and a coloring problem with costs
MPI-I-98-1-005	K. Jansen	The mutual exclusion scheduling problem for permutation and comparability graphs
MPI-I-98-1-004	S. Schirra	Robustness and Precision Issues in Geometric Computation
MPI-I-98-1-003	S. Schirra	Parameterized Implementations of Classical Planar Convex Hull Algorithms and Extreme Point Compuations
MPI-I-98-1-002	G.S. Brodal, M.C. Pinotti	Comparator Networks for Binary Heap Construction
MPI-I-98-1-001	T. Hagerup	Simpler and Faster Static AC^0 Dictionaries
MPI-I-97-2-012	L. Bachmair, H. Ganzinger, A. Voronkov	Elimination of Equality via Transformation with Ordering Constraints
MPI-I-97-2-011	L. Bachmair, H. Ganzinger	Strict Basic Superposition and Chaining
MPI-I-97-2-010	S. Vorobyov, A. Voronkov	Complexity of Nonrecursive Logic Programs with Complex Values