Twin – A Design Pattern for Modeling Multiple Inheritance

Hanspeter Mössenböck

University of Linz, Institute of Practical Computer Science, A-4040 Linz
moessenboeck@ssw.uni-linz.ac.at

Abstract. We introduce an object-oriented design pattern called *Twin* that allows us to model multiple inheritance in programming languages that do not support this feature (e.g. Java, Modula-3, Oberon-2). The pattern avoids many of the problems of multiple inheritance while keeping most of its benefits. The structure of this paper corresponds to the form of the design pattern catalogue in [GHJV95].

1. Motivation

Design patterns are schematic standard solutions to recurring software design problems. They encapsulate a designer's experience and makes it reusable in similar contexts. Recently, a great number of design patterns has been discovered and published ([GHJV95], [Pree95], [BMRSS96]). Some of them are directly supported in a programming language (e.g. the *Prototype* pattern in Self or the *Iterator* pattern CLU), some are not. In this paper we describe a design pattern, which allows a programmer to simulate multiple inheritance in languages which do not support this feature directly.

Multiple inheritance allows one to inherit data and code from more than one base class. It is a controversial feature that is claimed to be indispensable by some programmers, but also blamed for problems by others, since it can lead to name clashes, complexity and inefficiency. In most cases, software architectures become cleaner and simpler when multiple inheritance is avoided, but there are also situations where this feature is really needed. If one is programming in a language that does not support multiple inheritance (e.g. in Java, Modula-3 oder Oberon-2), but if one really needs this feature, one has to find a work-around. The *Twin* pattern—introduced in this paper—provides a standard solution for such cases. It gives one most of the benefits of multiple inheritance while avoiding many of its problems.

The rest of this paper is structured according to the pattern catalogue in [GHJV95] so that the Twin pattern could in principle be incorporated into this catalogue.

1.1 Example

As a motivating example for a situation that requires multiple inheritance, consider a computer ball game consisting of active and passive game objects. The active objects

are balls that move across the screen at a certain speed. The passive objects are paddles, walls and other obstacles that are either fixed at a certain screen position or can be moved under the control of the user.

The design of such a game is shown in Fig. 1. All game items (paddles, walls, balls, etc.) are derived from a common base class *GameItem* from which they inherit methods for drawing or collision checking. Methods such as *draw()* and *intersects()* are abstract and have to be refined in subclasses. *check()* is a template method, i.e. it consists of calls to other abstract methods that must be implemented by concrete game item classes later. It tests if an item intersects with some other and calls the other item's *collideWith()* method in that case. In addition to being game items, active objects (i.e. balls) are also derived from class *Thread*. All threads are controlled by a scheduler using preemptive multitasking.

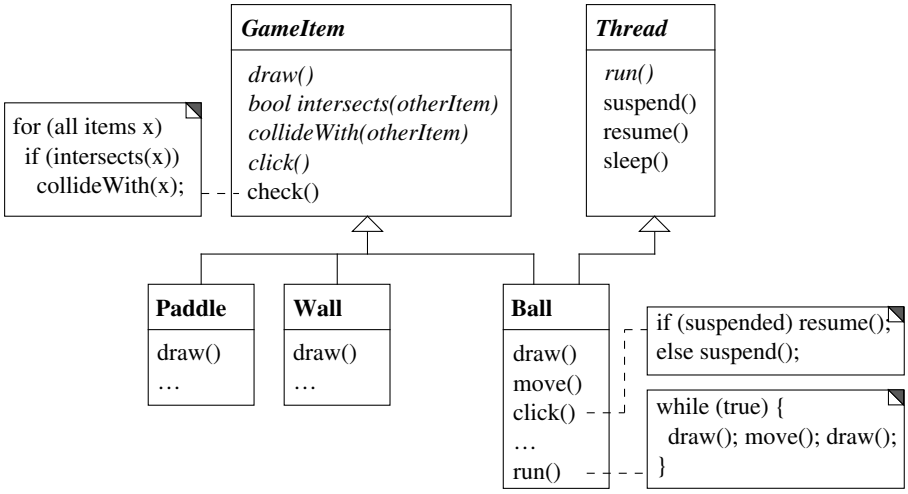

Fig. 1. Class hierarchy of a computer ball game

The body of a ball thread is implemented in its *run()* method. When a ball thread is running, it repeatedly moves and draws the ball. If the user clicks on a ball, the ball sends itself a *suspend()* message to stop its movement. Clicking on the ball again sends a *resume()* message to make the ball moving again.

The important thing about this example is that balls are *both game items and threads* (i.e. they are compatible with both). They can be linked into a list of game items, for example, so that they can be sent *draw()* and *intersects()* messages. But they can also be linked into a list of threads from which the scheduler selects the next thread to run. Thus, balls have to be compatible with both base classes. This is a typical case where multiple inheritance is useful.

Languages like Java don't support multiple inheritance, so how can we implement this design in Java? In Java, a class can extend only one base class but it can implement several interfaces. Let's see, if we can get along with multiple interface inheritance here. *Ball* could extend *Thread* and thus inherit the code of *suspend()* and *resume()*. However, it is not possible to treat *GameItem* just as an interface because *GameItem* is not fully abstract. It has a method *check()*, which contains code. *Ball*

would like to inherit this code from *GameItem* and should therefore extend it as well. Thus *Ball* really has to *extend* two base classes.

This is the place where the Twin pattern comes in. The basic idea is as follows: Instead of having a single class *Ball* that is derived from both *GameItem* and *Thread*, we have two separate classes *BallItem* and *BallThread*, which are derived from *GameItem* and *Thread*, respectively (Fig. 2). *BallItem* and *BallThread* are closely coupled via fields so that we can view them as a Twin object having two ends: The *BallItem* end is compatible with *GameItem* and can be linked into a list of game items; the *BallThread* end is compatible with *Thread* and can be linked into a list of threads.

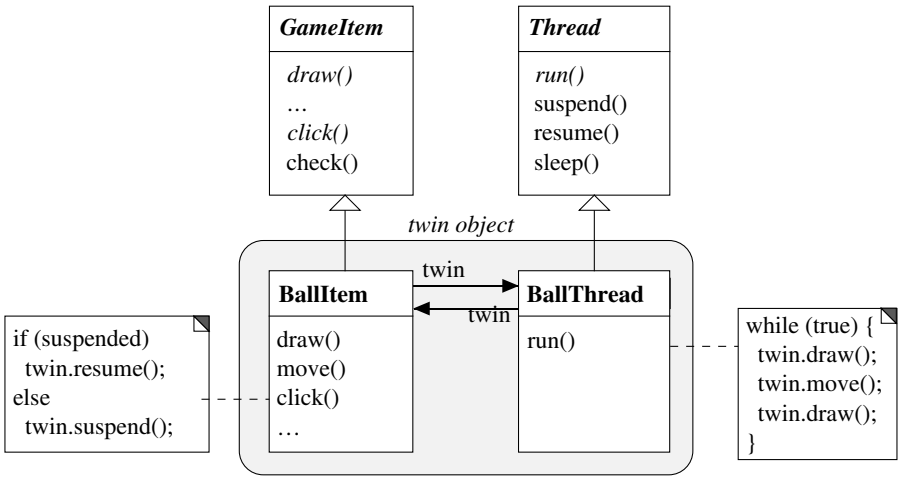

Fig. 2. The class *Ball* from Fig.1 was split into two classes, which make up a twin object

Twin objects are always created in pairs. When the scheduler activates a *BallThread* object by calling its method *run()*, the object moves the ball by sending its twin the messages *move()* and *draw()*. On the other hand, when the user clicks on a ball with the mouse, the *BallItem* object reacts to the click and sends its twin the messages *suspend()* and *resume()* as appropriate.

Using only single inheritance, we have obtained most of the benefits of multiple inheritance: Active game objects inherit code from both *GameItem* and *Thread*. They are also compatible with both, i.e. they can be treated both as game items (*draw*, *click*) and as threads (*run*). As a pleasant side effect, we have avoided a major problem of multiple inheritance, namely name clashes. If *GameItem* and *Thread* had fields or methods with the same name, they would be inherited by *BallItem* and *BallThread* independently. No name clash would occur. Similarly, if *GameItem* and *Thread* had a common base class *B*, the fields and methods of *B* would be handed down to *BallItem* and to *BallThread* separately—again without name clashes.

2. Applicability

The Twin pattern can be used

- to simulate multiple inheritance in a language that does not support this feature.
- to avoid certain problems of multiple inheritance such as name clashes.

3. Structure

The typical structure of multiple inheritance is described in Fig.3.

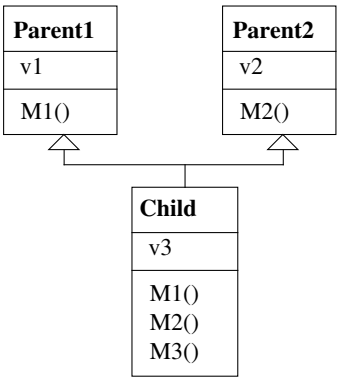

Fig. 3. Typical structure of multiple inheritance

It can be replaced by the Twin pattern structure described in Fig.4.

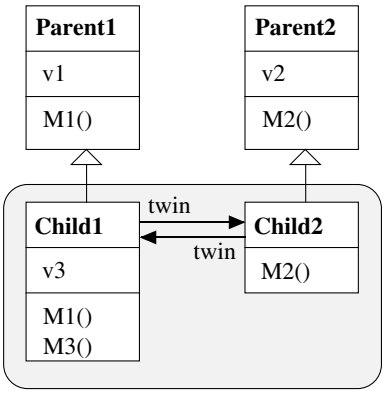

Fig. 4. Typical structure of the Twin pattern

4. Participants

Parent1 (*GameItem*) and **Parent2** (*Thread*)

- The classes from which you want to inherit.

Child1 (*BallItem*) and **Child2** (*BallThread*)

- The subclasses of *Parent1* and *Parent2*. They are mutually linked via fields. Each subclass may override methods inherited from its parent. New methods and fields are usually declared just in one of the subclasses (e.g. in *Child1*).

5. Collaborations

- Every child class is responsible for the protocol inherited from its parent. It handles messages from this protocol and forwards other messages to its partner class.
- Clients of the twin pattern reference one of the twin objects directly (e.g. *ballItem*) and the other via its twin field (e.g. *ballItem.twin*).
- Clients that rely on the protocols of *Parent1* or *Parent2* communicate with objects of the respective child class (*Child1* or *Child2*).

6. Consequences

Although the Twin pattern is able to simulate multiple inheritance, it is not identical to it. There are several problems that one has to be aware of:

1. *Subclassing the Twin pattern*. If the twin pattern should again be subclassed, it is often sufficient to subclass just one of the partners, for example *Child1*. In order to pass the interface of both partner classes down to the subclass, it is convenient to collect the methods of both partners in one class. One can add the methods of *Child2* also to *Child1* and let them forward requests to the other partner (Fig.5).

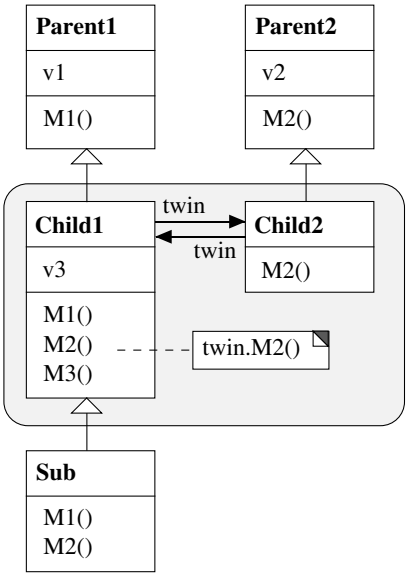

Fig. 5. Subclassing a twin class. *Child1.M2()* forwards the message to *Child2.M2()*

This solution has the problem that *Sub* is only compatible with *Child1* but not with *Child2*. If one wants to make the subclass compatible with both *Child1* and *Child2* one has to model it according to the Twin pattern again (Fig.6).

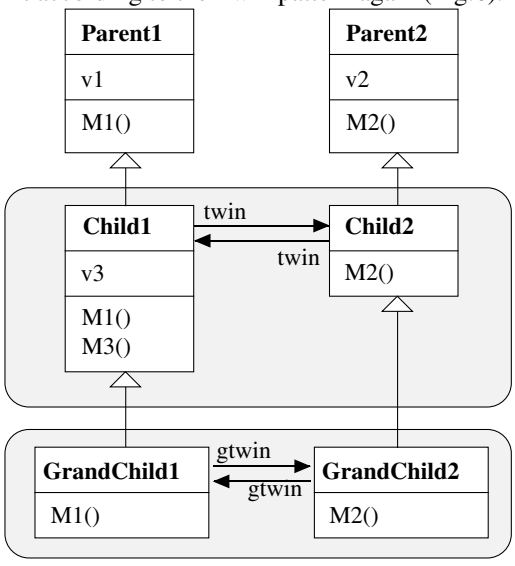

Fig. 6. The subclass of *Child1* and *Child2* is again a Twin class

2. *More than two parent classes*. The Twin pattern can be extended to more than two parent classes in a straightforward way. For every parent class there must be a child class. All child classes have to be mutually linked via fields (Fig.7).

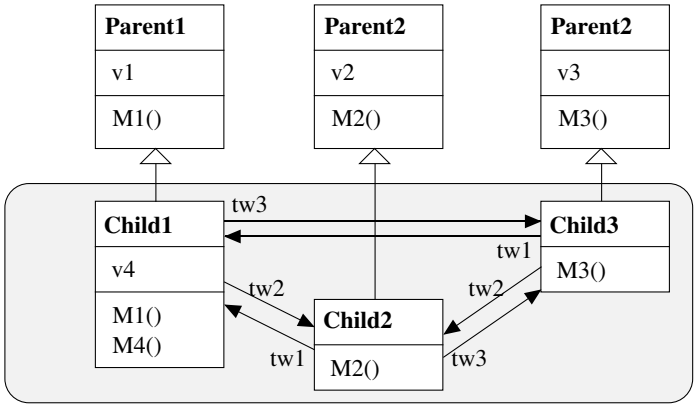

Fig. 7. A Twin class derived from three parent classes

Although this is considerably more complex than multiple inheritance, it is rare that a class inherits from more than two parent classes.

7. Implementation

The following issues should be considered when implementing the Twin pattern:

1. *Data abstraction*. The partners of a twin class have to cooperate closely. They probably have to access each others' private fields and methods. Most languages provide features to do that, i.e. to let related classes see more about each other than foreign classes. In Java, one can put the partner classes into a common package and implement the private fields and methods with the *package* visibility attribute. In Modula-3 and Oberon one can put the partner classes into the same module so that they have unrestricted access to each others' components.

2. *Efficiency*. The Twin pattern replaces inheritance relationships by composition. This requires forwarding of messages, which is less efficient than inheritance. However, multiple inheritance is anyway slightly less efficient than single inheritance [Str89] so that the additional run time costs of the Twin pattern are not a major problem.

8. Sample Code

We sketch the implementation of the motivating example (a computer game board with moving balls) in Java. The board is represented by a class *GameBoard*. It has a certain width and height and a reference to a list of game items.

```
public class Gameboard extends Canvas {
  public int width, height;
  public GameItem firstItem;
  …
}
```

The game items are derived from an abstract class *GameItem*. Every item has a reference to the game board, a position on this board and a reference to the next game item. It has abstract methods to draw itself, to react on mouse clicks, to check whether it intersects with some other game item and to take measures for a collision with other game items.

```
public abstract class GameItem {
  Gameboard board;
  int posX, posY;
  GameItem next;
  public abstract void draw();
  public abstract void click (MouseEvent e);
  public abstract boolean intersects (GameItem other);
  public abstract void collideWith (GameItem other);
  public void check() { … }
}
```

The method *check()* is a template method, which checks if this object intersects with any other object on the board. If so, it does whatever it has to do for a collision.

```
public void check() {
  GameItem x;
  for (x = board.firstItem; x != null; x = x.next)
    if (intersects(x)) collideWith(x);
}
```

Balls are twin objects derived from *GameItem* and *Thread*. As shown in Fig. 2 we implement the twin group as *BallItem* (a subclass of *GameItem*) and *BallThread* (a subclass of *Thread*). Ball items move at a certain speed (*dx*, *dy*) and have to override the inherited methods *draw*, *click*, *intersects* and *collideWith*.

```
public class BallItem extends GameItem {
  BallThread twin;
  int radius;
  int dx, dy;
  boolean suspended;
  public void draw() {
    board.getGraphics().drawOval(posX-radius,
    posY-radius, 2*radius, 2*radius); }
  public void move() { posX += dx; posY += dy; }
  public void click() {…}
  public boolean intersects (GameItem other) {…}
  public void collideWith (GameItem other) {…}
}
```

In order to simplify things, we assume that balls can only collide with walls, which are another kind of game items. The *intersects* method of a *BallItem* can then be implemented as

```
public boolean intersects (GameItem other) {
  if (other instanceof Wall)
    return posX - radius <= other.posX
      && other.posX <= posX + radius
    || posY - radius <= other.posY
      && other.posY <= posY + radius;
  else return false;
}
```

A collision with a wall changes the direction of the ball, which can be implemented as

```
public void collideWith (GameItem other) {
  Wall wall = (Wall) other;
  if (wall.isVertical) dx = - dx; else dy = - dy;
}
```

When the user clicks on a moving ball it stops; when he clicks on a stopped ball it starts to move again. This is implemented by suspending and resuming the corresponding ball thread (the twin object).

```
public void click() {
  if (suspended) twin.resume(); else twin.suspend();
  suspended = ! suspended;
}
```

The class *BallThread* is derived from the standard class *java.lang.Thread*. It has a reference to its twin class *BallItem*. The only method that has to be implemented is *run()*. The implementation of other methods such as *suspend()* and *resume()* is inherited from *Thread*.

```
public class BallThread extends Thread {
  BallItem twin;
  public void run() {
    while (true) {
      twin.draw(); /*erase*/ twin.move(); twin.draw();
    }
  }
}
```

When a new ball is needed, the program has to create both a *BallItem* and a *BallThread* object and link them together, for example:

```
public static BallItem newBall
(int posX, int posY, int radius) {//method of GameBoard
  BallItem ballItem = new BallItem(posX, posY, radius);
  BallThread ballThread = new BallThread();
  ballItem.twin = ballThread;
  ballThread.twin = ballItem;
  return ballItem;
}
```

The returned ball item can be linked into the list of game items in the game board. The corresponding ball thread can be started to make the ball move.

9. Known Uses

The motivating example of a ball game (Section 1) was implemented as a teaching exercise in Oberon-2, a language that does not support multiple inheritance. The Oberon system uses cooperative multitasking. It maintains a list of user processes that are activated whenever the system is idle. A ball is a special instance of a process and at the same time a game object.

Another example can be found in the context of Java applets. Applets are active objects that live on Web pages and react on user input such as mouse clicks. When a user clicks on an applet, the applet notifies all registered mouse listeners to react on the event. If an applet wants to react on the click itself, it has to implement the *MouseListener* interface, so that it can be registered as an appropriate listener with itself. It must also extend the class *Applet*. The following code shows the declaration of a class *MyApplet*:

```
class MyApplet extends Applet implements MouseListener{
  …
}
```

The *MouseListener* interface (a standard interface of the Java libraries) specifies 5 methods that have to be implemented in *MyApplet*:

```
interface MouseListener extends EventListener {
  public void mousePressed (MouseEvent event);
  public void mouseClicked (MouseEvent event);
  public void mouseReleased (MouseEvent event);
  public void mouseEntered (MouseEvent event);
  public void mouseExited (MouseEvent event);
}
```

Some of these methods are often identical in different listener implementations. For example, several listeners change the shape of the cursor in the same way when it enters or exits the applet area on the screen. Therefore, we would like to have a prefabricated mouse listener class (*StdMouseListener*), which already provides standard implementations for the methods *mouseEntered* and *mouseExited*. Other listeners could then inherit these standard implementations.

We are now in a situation where we would like to inherit code from *two* classes, namely from *Applet* and *StdMouseListener*, but this is not possible in Java. We can only inherit from one class. We can, however, apply the Twin pattern, which results in the following architecture (Fig.8).

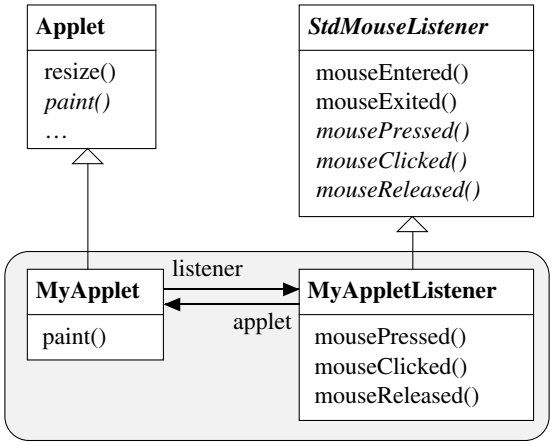

Fig. 8. A twin applet that inherits code both from *Applet* and from *StdMouseListener*

MyApplet inherits code from *Applet*; *MyAppletListener* inherits code from *StdMouse-Listener*. A *MyAppletListener* object will be registered as a mouse listener for *MyApplet*. When it is notified about a mouse click it accesses its applet to perform an appropriate action.

In [CaW98] a similar solution is presented using inner classes. *MyAppletListener* is implemented there as an inner class of *MyApplet*. This allows *MyAppletListener* to access all private instance variables of *MyApplet*. No explicit link between the classes is necessary. However, this solution is asymmetric. *MyApplet* cannot access the private instance variables of *MyAppletListener*.

10. Related Patterns

The Twin pattern is related to the *Adapter* pattern, especially to the *Two-Way-Adapter* described in [GHJV95], which is recommended when two different clients need to view an object differently. However, the *Two-Way-Adapter* is implemented with multiple inheritance while the Twin avoids this feature.

Acknowledgements

The technique described in this paper was discovered by Robert Griesemer in the implementation of a game program in Oberon. It was also described—although not as a design pattern—in [Tem93] and [Moe93].

References

[BMRSS96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: Pattern-oriented Software Architecture: A System of Patterns. Wiley 1996.

[CaW98] Campione M., Walrath K.: The Java Tutorial, 2nd edition, Addison-Wesley, 1998.

[GHJV95] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley 1995.

[Moe93] Mössenböck H.: Objektorientierte Programmierung in Oberon-2. Springer-Verlag 1993.

[Pree95] Pree W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley 1995.

[Str89] Stroustrup B.: Multiple Inheritance for C++. Proceedings EUUG Spring Conference, Helsinki, May 1989.

[Tem93] Templ J.: A Systematic Approach to Multiple Inheritance Implementation. SIGPLAN Notices 28 (4): 61-66