Design Patterns in Oberon-2 and Component Pascal

translated excerpts from

Objektorientierte Programmierung in Oberon-2

by

Hanspeter Mössenböck

English translation: **Bernhard Treutwein**

Authorized translation of Chapter 9 "Entwurfsmuster" of Hanspeter Mössenböck: Objektorientierte Programmierung in Oberon-2, 3rd Edition, 1998.
Permission granted by Springer, Heidelberg.

9 Design Patterns

Experienced programmers distinguish themselves by having a pool of solutions for recurring problems. This specifies their expert knowledge. If they are assigned a certain task, they do not have to work out the solution at hand, but they can refer to a proven solution from their experience.

Such standard solutions are called *design patterns*. They provide schematic solutions for common recurring problems. In the world of object oriented programming, design patterns were introduced by the excellent book of Gamma et al. [GHJV96] (see also [Pre95] and [BMRSS96]). Design patterns are neither restricted to nor an invention of object oriented programming. Design patterns are nothing other than the algorithms and data structures of object oriented programming. Object-structures would be a well-suited synonym.

9.1 Motivation

Patterns in Conventional Programs

We want to look briefly into the properties and advantages of design patterns before reviewing a collection of useful patterns. We choose a well known example from conventional programming: The structure of a binary tree shown in Fig. 9.1. It represents a typical *pattern.* Binary trees are known for being efficient in searching large data sets. If you want an efficient search, a binary tree is a proven way to solve this task.

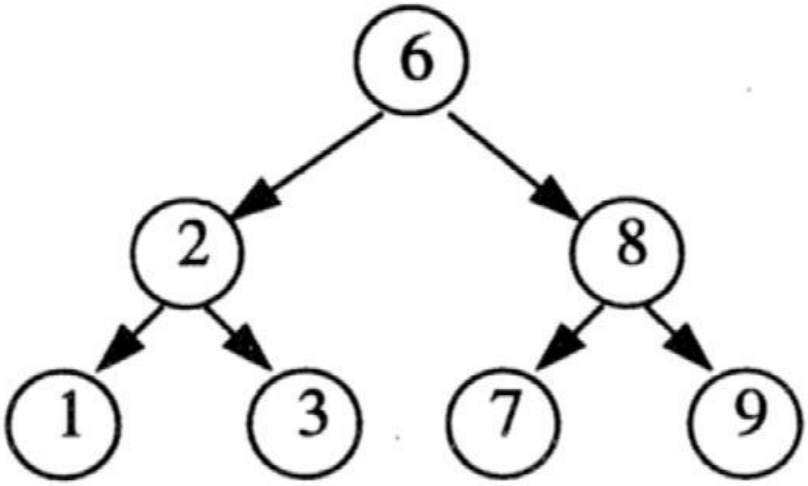


Fig. 9.1: A binary tree as an example of a design pattern

Constituents of a Design Pattern

What are the constituents of a pattern? Essentially it has a ***name***, a ***description of the problem***, and a ***solution***.

The ***name*** presents a handy and familiar terminology, which can be used by developers for communication. It is easier to say "take a binary tree" than "setup your elements in such a way that every element has up to two children and the value of its left child is less than or equal to the value of its right child".

The ***description of the problem*** for the binary tree could be: Use a

binary tree, if you have a large data set in main memory and you want to efficiently insert, delete, and search it. The description of the problem may hint at restrictions and dangers. For example, binary trees are not well suited if the data changes frequently. In this case there is a danger that the tree will degenerate.

The third component of a pattern is its solution or ***implementation***, which is quite often deliberately abstract and does not dive into the details. This makes the pattern independent of specific implementation languages or concrete data structures. The realization of a binary tree depends solely on the relations defined between the nodes and children of the tree.

Patterns in Object-Oriented Programs

Design patterns in the ***object-oriented*** meaning describe the cooperation of classes, and/or objects, when working on the solution of a particular problem. They are therefore most often described with the help of class-diagrams. Since it is frequently necessary to sketch the dynamic cooperation of the classes, the class-diagrams are augmented with code fragments or a description of their behavior via interaction-diagrams.

Gamma et al. (1995) gave a catalogue of some 25 design patterns. Here we want to mention only the more important ones, since some patterns are quite similar. We also want to show some other patterns, which are not included in Gamma et al., that appear quite useful.

Gamma et al. separate the patterns into three main categories: ***Creational patterns*** are about the flexible creation of objects and/or object-structures (classes); ***Structural patterns*** are about frequently occurring aggregations of objects to larger structures; ***Behavioral patterns*** finally are about frequently occurring dynamic behavior of objects.

9.2 Creational Patterns

Objects are almost always allocated dynamically. For this purpose the standard procedure NEW exists in Oberon-2. Other programming languages offer similar constructors. But the simple creation is frequently not that simple. Sometimes it is necessary to have additional actions (e.g., initialization), often it is a-priori unknown, which type the object will be. In this chapter we will introduce three creational patterns, which can handle such situations:

- Constructor Creation and initialization of objects
- Factory Creation of object with a variable type
- Prototype Creation of object with a variable type

9.2.1 Constructor

When creating an object it is desirable to initialize its fields. If you do not want to forget the initialization, it is reasonable to combine creation and initialization into a single action. This is the purpose of the ***constructor pattern.***

Many programming languages (e.g. C++ or Java) have contructors already as a feature of the language. Oberon-2 and Component Pascal do not have constructors, but they can be easily modelled with the constructor pattern. Component Pascal additionally offers the record attribute LIMITED, which explicitely forbids allocation, with NEW, outside the declaring module.

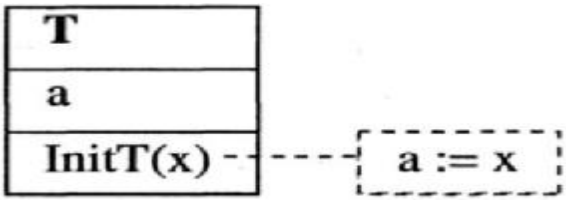


Fig. 9.2 A class with a method for initialization

Let us consider a class *T* with an attribute *a* and a method *InitT*, which initializes *a* (Fig. 9.2). A constructor for *T* will be implemented as a function-procedure *NewT*, which creates an object of type *T* and calls the initialization-procedure *InitT*:

```
TYPE
   T* = POINTER TO RECORD
      a- : REAL
   END;

PROCEDURE ( t : T ) InitT* ( x  : REAL );
   BEGIN
      t.a := x
END InitT;

PROCEDURE NewT* ( x : REAL ) : T;
   VAR t : T;
   BEGIN
      NEW(t); t.InitT(x);
      RETURN t
END NewT;
```

Now whenever a *T*-object is needed, it can be created and initialized with a call to its constructor:

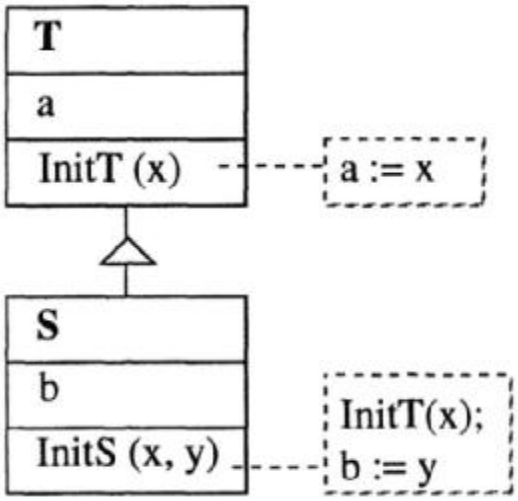
```
obj := NewT(x)
```

This pattern is easily applied to arbitrary classes by substituting *T* with the particular class-name (e.g. *NewRectangle* and *InitRectangle*).

If an object of a subclass is created, it is not only necessary to ??? initialize both, the attributes of this (sub)class, and the attributes of the baseclass(es). The constructor-pattern can be modified to suite this purpose. *S* being a subclass of *T*, a constructor *NewS* is implemented:

```
obj := NewS(x,y)
```

which creates a new *S* object. This new *S* object is initialzed by a call to *InitS,* which in turn initializes the baseclass attributes by a call to *InitT*.




```
TYPE
   S* = POINTER TO RECORD(T)
      b- : REAL
   END;

PROCEDURE ( s: S )InitS* ( x, y  : REAL );
   BEGIN
      s.b := x; s.InitT(x)
END InitS;

PROCEDURE NewS* ( x, y : REAL ) : S;
   VAR s : S;
   BEGIN
      NEW(s); s.InitS(x, y);
      RETURN s
END NewS;
```

Fig. 9.3 Constructor of a base class

The compilable example code can be found here: Constructor

9.2.2 Factory

The purpose of the factory design pattern is the creation of objects, where the dynamic type of the object is not a priori fixed (statically determined) in the program.

This is best seen in the following example. Let us assume that there exist different kinds of texts, which are derived from an abstract class *Text* (Fig. 9.4). *SimpleText* are simple ASCII-texts and *StyledText* can also have different fonts and styles.

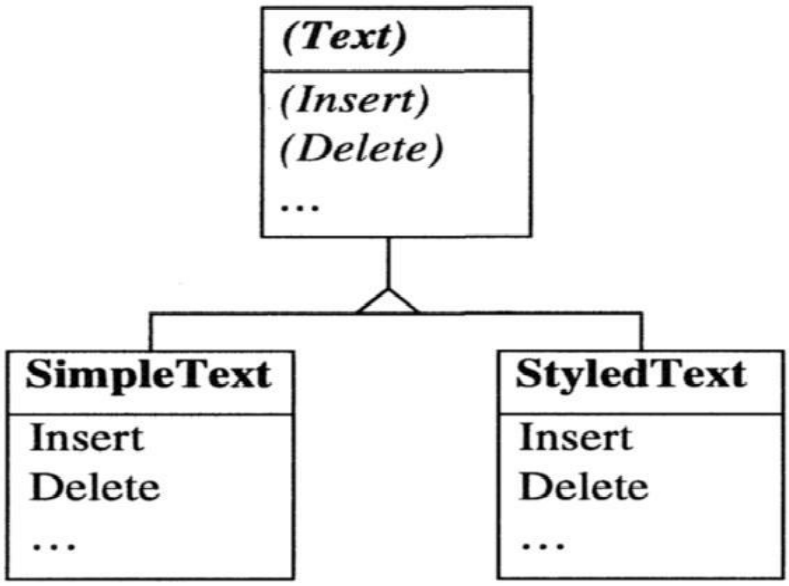


Fig. 9.4 Different kinds of texts

Let us now imagine that an editor is supposed to work either with *SimpleText* or *StyledText* at the decision of the user. Therefore the editor has an attribute *t* of the abstract class *Text*, which can hold either *a SimpleText* or a *StyledText* object (Fig. 9.5).

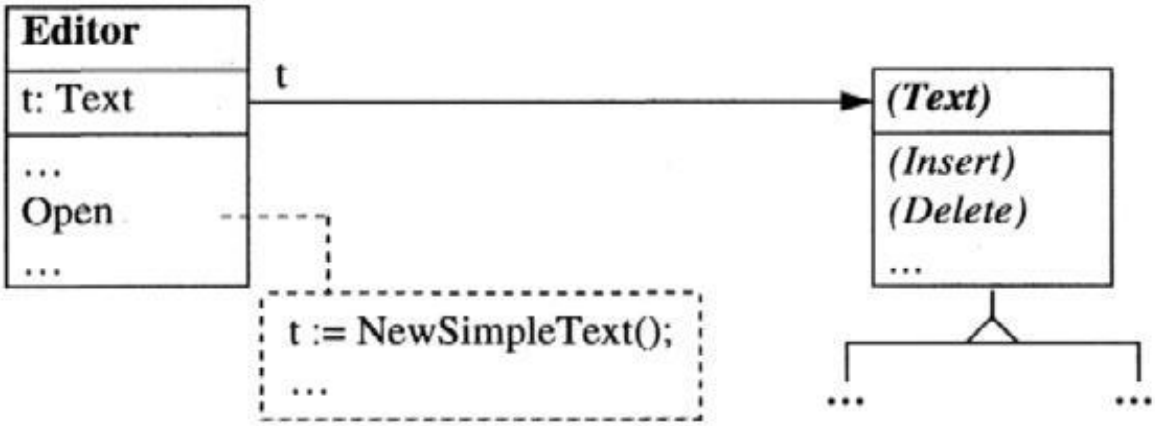


Fig. 9.5 The dynamic type of *t* is fixed statically

A new *Text*-object is allocated when an editor window with the method *Open* is opened. The editor must decide which kind of object will be created. In Fig. 9.5 the editor creates a *SimpleText*-object. This decision is burned into the code without any possibility to change it. If you want to create *StyledText*-objects, you have to change the code.

This problem can be solved with the *factory* design-pattern. Instead of *statically* fixing the type of the text, the generation of the text is left over to a a *factory-object*. For every kind of text, there is a special factory (*SimpleFactory, StyledFactory*, etc.), which generates the corresponding *Text*-object. All factories are derived from an abstract factory-class leading to the pattern in Fig. 9.6.

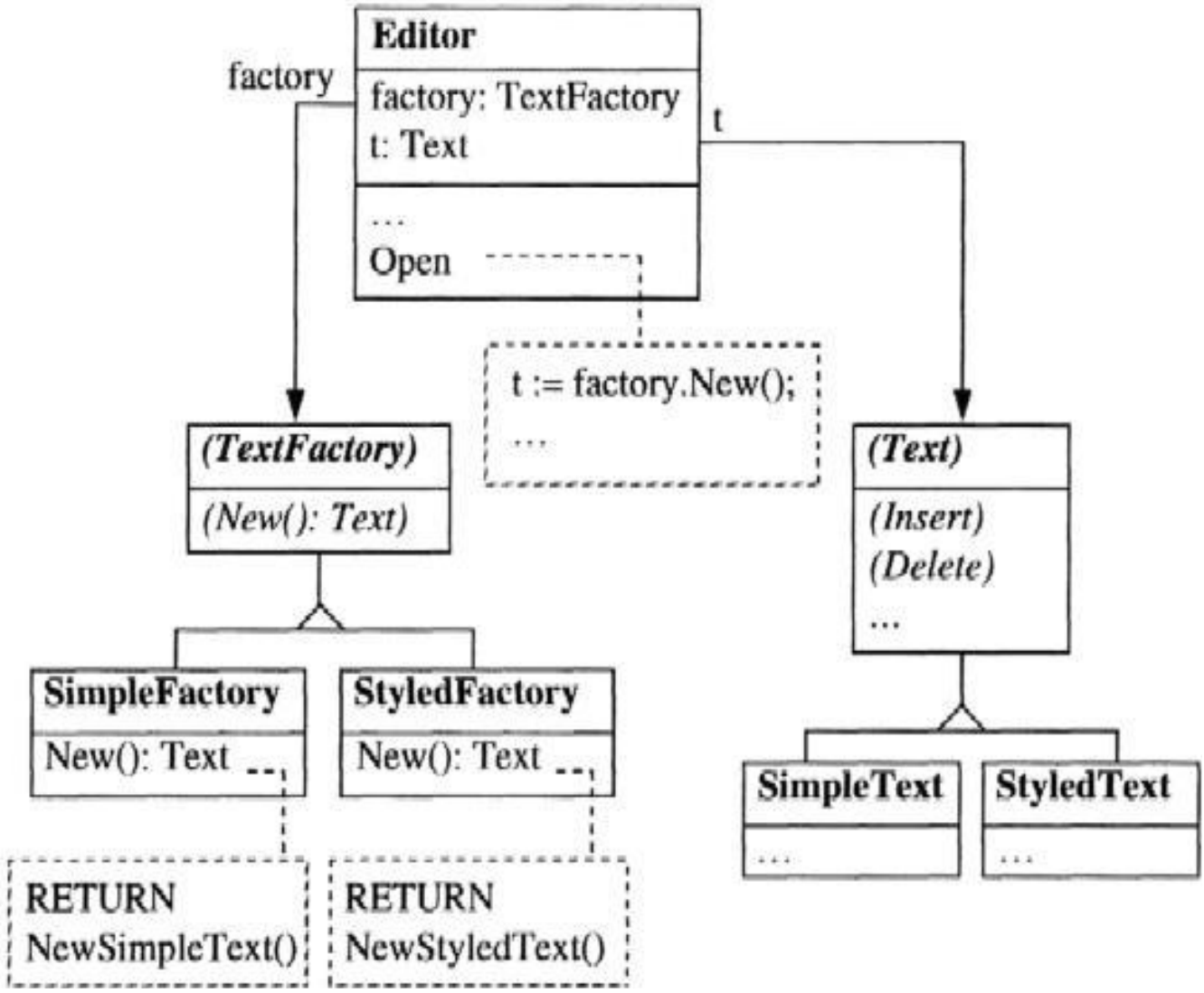


Fig. 9.6 Generation of a text by a factory-object

When the editor is initialized, a *SimpleFactory* or a *StyledFactory* object is assigned to the attribute *factory* of the editor. If a *StyledFactory* object is chosen, the call of the method *Open* yields a call of the *New*-method of the *StyledFactory.* Therefore a *StyledText*-

object is generated and returned to the editor.

The code of *Open* does not nail down which kind of text is generated. The text is produced by a factory, which can be chosen during initialization of the editor and can later be changed during runtime.

Summarizing: if objects have to be generated whose dynamic type should be flexible, they are not directly allocated but are requested from a factory-object. There can be different kinds of factories which generate different objects. The desired type of the factory is chosen during the initialization of the system.

9.2.3 Prototype

The purpose of the prototype-pattern is similar to the factory-pattern. It is used to create objects, when it is necessary to be flexible regarding their dynamic type. The concrete realization of a prototype is quite different compared to a factory and is simpler in many aspects.

If you need a new object of a certain type, it is not created on its own, but it is copied from a *prototype-object*, which already has the desired type.

Let us stick to the editor example from Chap. 9.2.2. In this case there are different kinds of texts, which can be used interchangeably in the editor. For each kind of text a prototype-object is created (i.e. a *SimpleText*-object and a *StyledText*-object). One of these prototype-objects is stored during the initialization of the editor in the attribute *protoText*. In the *Open*-method of the editor a copy of *protoText* is created and this copy is used as text in the editor's window.

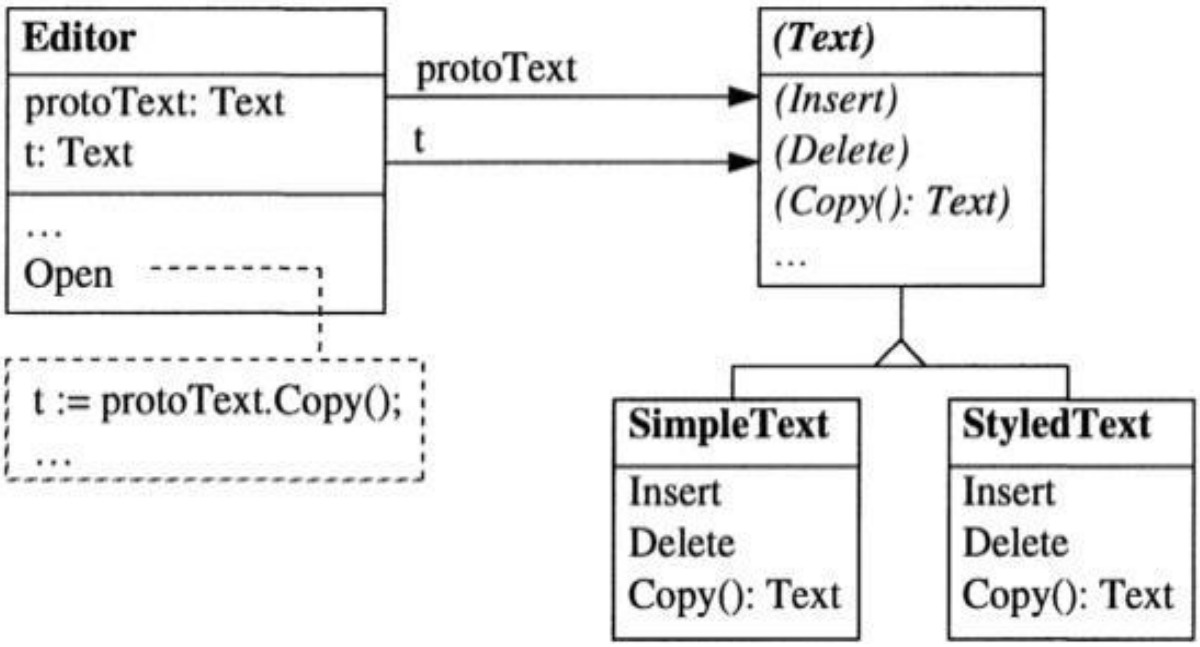


Fig. 9.7 Generation of a text by copying a prototype-object

What is the difference between the prototype-pattern and the factory-patter? The prototype-pattern requires the candidate objects (here the *Text*-objects) to clone themselves, i.e. they have a Copy-method. This does not present a large restriction, but may be unfeasible for an existing class-hierarchy. The factory-pattern does not have this restriction and can do a suitable initialization of the newly created ??? (well-suited) object, which may depend on the current state of the program. The drawback of the factory-pattern is that it requires a complete class-hierarchy for factories, which complicates the whole system.

Another advantage of the prototype-pattern is that it is not only possible to create a copy of a *single* object, but also a copy of a complete *subsystem* consisting of multiple objects. Such subsystems can be assembled during runtime and each *Copy*-operation can yield a copy of the subsystem as a whole. This might be difficult with a factory-object.

9.3 Structural Patterns

Objects rarely exist in isolation, but collaborate with other objects to solve a given task. For that purpose they build certain structures, which often occur in this special form. These forms will be described here under the category of structural patterns. We restrict ourselves here to the following patterns:

- Family Building up class-hierarchies
- Adapter Fitting a foreign class to a family
- Composite Aggregation of parts to a new part
- Decorator Serial connection of functionality
- Twin Avoiding multiple inheritance

9.3.1 Family

The family is a very simple - almost trivial - pattern. We mention it here just to introduce a name for this pattern. A family consists of an abstract class and its subclasses. Fig. 9.8 shows a family of GUI-objects and Fig. 9.4 is a family of text objects.

All members of a family have the same interface as its abstract base-class. Therefore the particular members of a family are exchangeable with each other. A program, which works with GUI-objects, can also work with *Checkbox, Button* and *Scrollbar.*

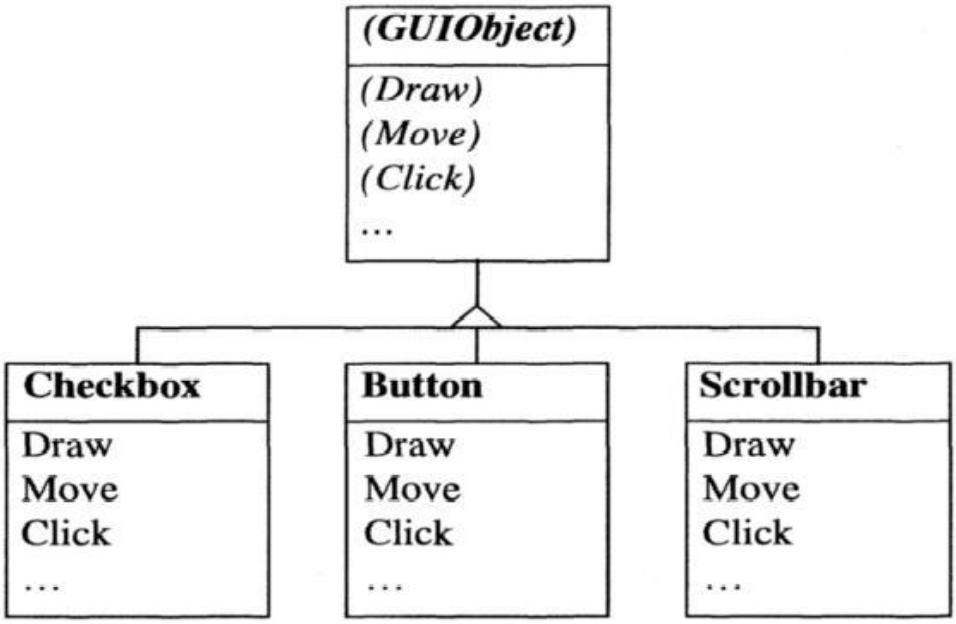


Fig. 9.8 Family of GUI-objects

9.3.2 Adapter

Sometimes you want to make an existing class compatible with a special family. In other words: you want to look at that class as a member of that family. For this purpose it would be necessary to derive it from the abstract base-class of the family. This is often impossible, because the class is derived from another base-class and you do not want, or you are not allowed to use, multiple inheritance. Moreover it may be possible that you do not have the source code and therefore you cannot change their inheritance relationships.

The solution for this problem is to introduce a new member of the family which works as an adapter to the foreign class. It implemements the messages of the family by translating the messages to the foreign class and redirecting them.

Let us consider an example: A graphic-editor manages a family of figures (lines, rectangles, circles, etc.). It should also be possible to work on texts in the graphic-editor. Let us assume that there exists a class *Text*, but it is not a member of the figure family. The latter implies that *Text* cannot be handled like a figure. Therefore we have to introduce a *TextAdapter*, which maps *Figure*-messages to *Text*-

messages (see Fig. 9.9). For example, the method *Draw* of the text-adaptor will be implemented by reading the text character by character and drawing the characters on the screen.

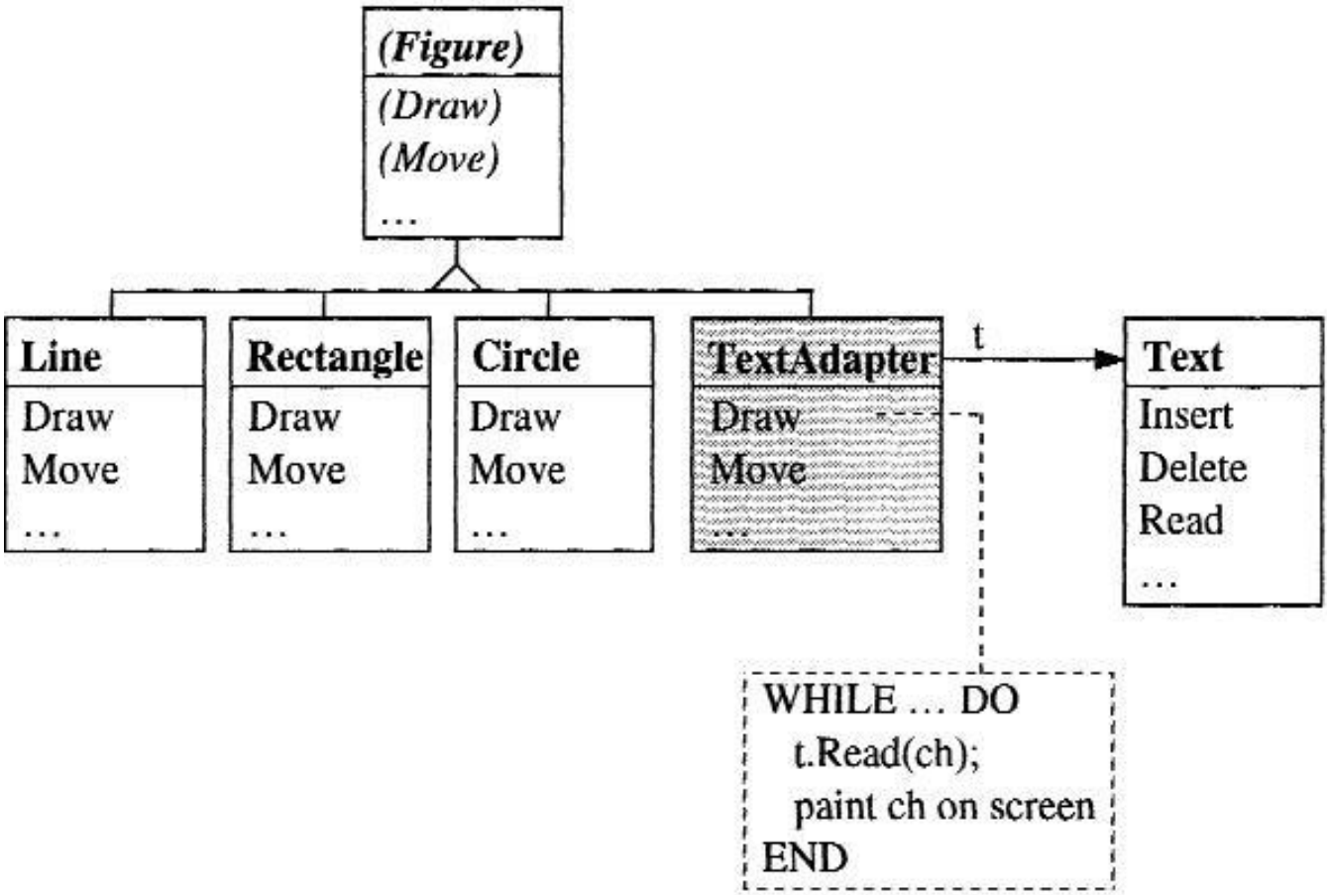


Fig. 9.9 Wrapping a *Text*-class into the *Figure-family* with the help of a *Text*-adaptor

A text-adaptor can also be considered as a tier, which encloses the *Text*-class and gives it a different interface. For that reason it is sometimes also called *Wrapper*.

The Adaptor design pattern is one of the most frequently used patterns. The problem of making two not yet related classes compatible with each other arises in almost all object-oriented programs. The Adaptor solves this problem quite easily.

9.3.2 Composite

It is often necessary to collect several single objects in a larger object, which behaves again as a single object, which in turn can be aggregated with other objects to an even larger object.

The structure, which results from a recursive grouping of single objects, is called *Composite*. It can be found e.g. in a graphic-editor, where several individual figures can be merged into a group, which can can be drawn, moved, or copied like a single part. Fig. 9.10 shows this situation.

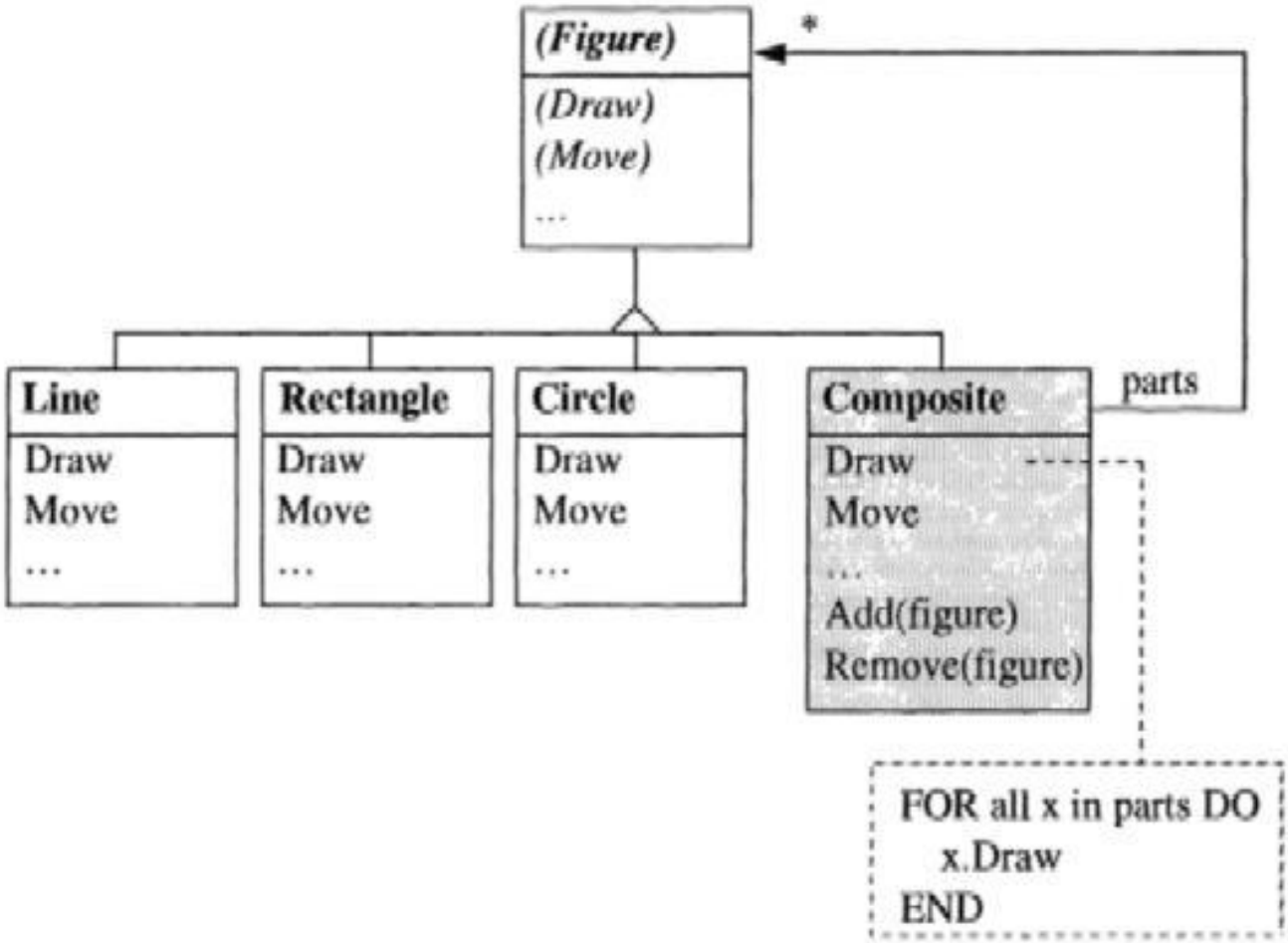


Fig. 9.10 Composite used for grouping of figures in a grafic-editor

Composite is a subclass of *Figure*, so it can be handled like a figure. It consists of a set of figures (*parts*), where these parts in turn can be themselves *Composites*. Messages sent to *Composites* are

mapped to messages to the sub-objects. A *Draw*-message to the group yields several *Draw*-messages to the parts. It is often useful to also have a pointer from the parts to the composite. This pointer is advantageously an attribute of *Figure*.

Composite has an *Add*- and a *Remove*-method for adding and removing parts to and from the group. In GoF [GHJV95] *Add* and *Remove* are methods of the abstract base class of *Composite*, but this appears to be a bit artificial, since both cannot be implemented meaningfully for single objects.

There are many applications for the composite-pattern. It is useful for window management, where single frames can have subframes. Yet another example is an editor for mathematical (or chemical) formulae, where a term (e.g. an integral) can consist of subterms (e.g. fractions) and these in turn have other terms as components.

9.3.4 Decorator

The *Decorator* pattern is used to add new behavior to a class without changing the class or subclassing it. The new behavior is *put in front of the class* and might be added or removed during runtime.

We choose an example from the GoF [GHJV95] for illustrating this pattern. A window system uses *Frames* as rectangular drawing-areas for text or graphics. There exists a *Frame*-family with subclasses like *TextFrame* or *GraphicFrame*. You want to add properties to these frames, e.g., you want to have scrollbars for moving the contents of the frame or you want to have a different style for the borders of the frame.

If these properties are realized by subclassing the class hierarchy it would be very complex due to the large number of possible combinations. Fig. 9.11 shows what happens, if frames with scrollbars and frames with borders are implemented as separate subclasses.

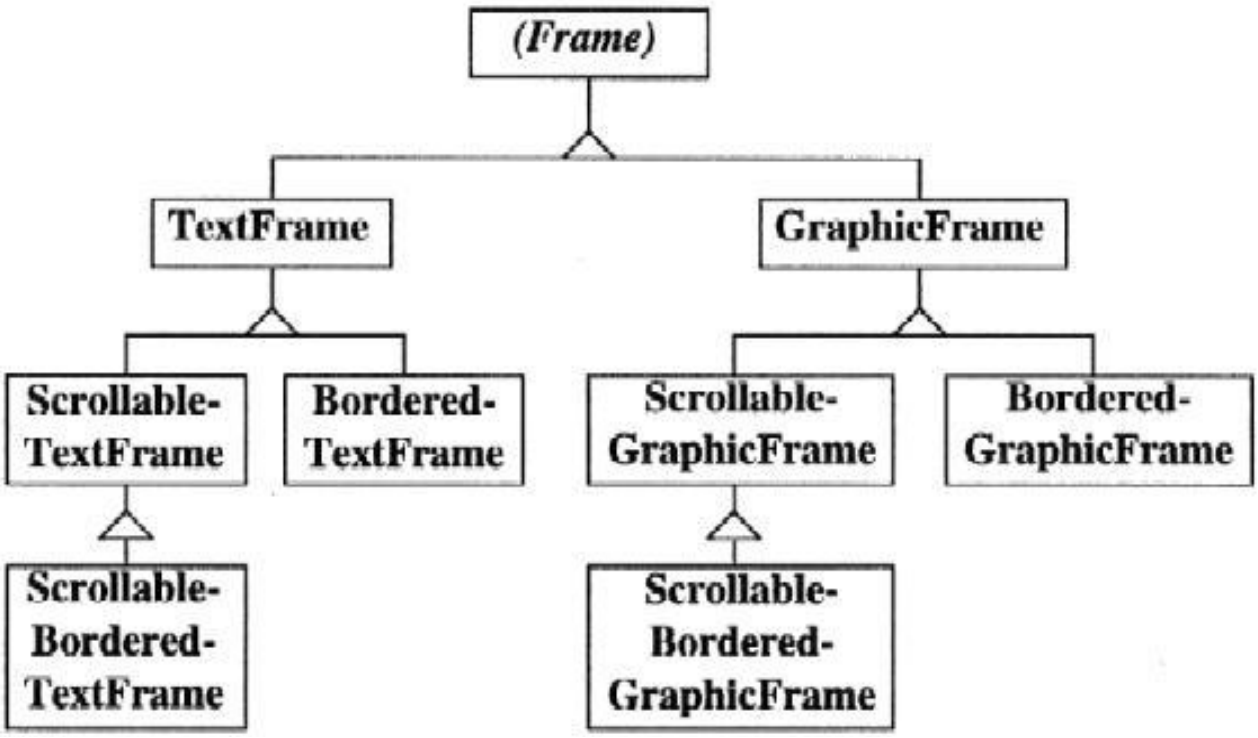


Fig. 9.11 Explosion of the number of classes if new properties are implemented as subclasses

The idea of the *Decorator*-pattern is to implement the properties with a class, which is *put in front* of the main class instead of subclassing it. Fig. 9.12 shows how this can be done for this example. In front of the *TextFrame* object there is a *ScrollDecorator* object. If this object receives a *Draw*-message, it paints a scrollbar and forwards the *Draw*-message to the *TextFrame*-object so that the latter can draw its text.

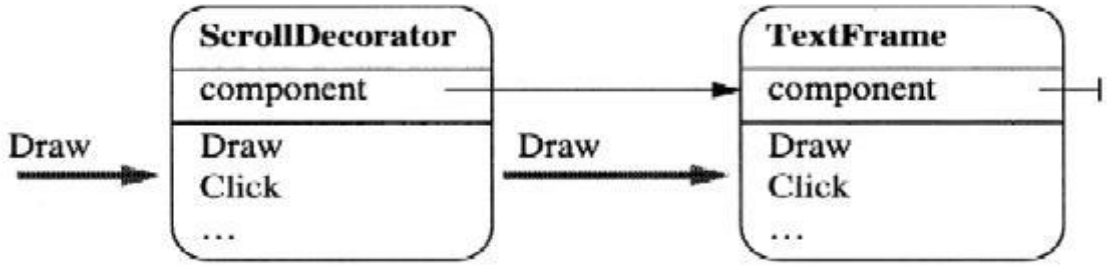


Fig. 9.12 *ScrollDecorator* object put in front of a *TextFrame*

All clients, which worked with a *TextFrame*-object must send their messages now to the *ScrollDecorator*-object. This works like a proxy

of the *TextFrame*-object and the *ScrollDecorator*-object must have the same interface, or, in other words, it must belong to the same family as the *TextFrame*-object. So we get the class-diagram shown in Fig. 9.13 for the decorator-pattern.

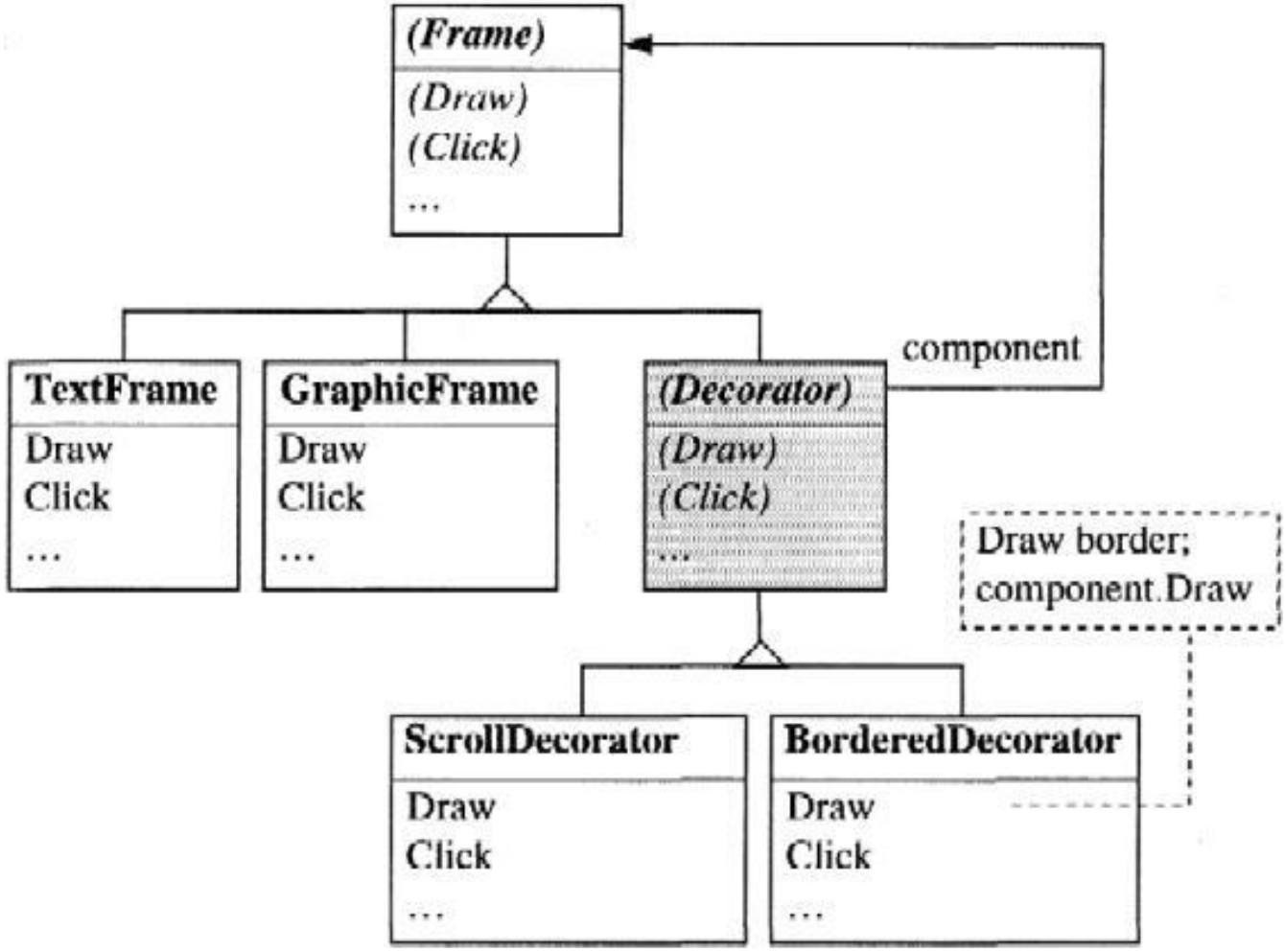


Fig. 9.13 *ScrollDecorator* object put in front of a *TextFrame*

The class *Decorator* belongs to the *Frame*-family and can therefore be handled as any other frame. With the attribute *component* a *Decorator* can be be connected to any other member of the *Frame*-family, especially with another *Decorator*.

Therefore it is possible to have before a *GraphicFrame* both a *BorderedDecorator* and a *ScrollDecorator*, in which case both properties are combined. When *BorderedDecorator* receives a *Draw*-message, it draws a border and forwards the *Draw*-message to the *ScrollDecorator*, which in turn draws a scrollbar and forwards the

Draw-message to the *GraphicFrame*-object.

You should note that not only can all properties be combined arbitrarily without creating an immense number of subclasses, but also that the combination may be changed during runtime. For example, it is possible to put borders on the frame only if the mouse pointer is over this frame. Inheritance is less flexible in this aspect. An inheritance relation cannot be changed during runtime.

The main problem of the decorator pattern is that the identity of the decorated object is lost. Clients do not refer to the *Frame*-object but to the *Decorator*-object. If it is dynamically connected upstream, you have to pay attention that all existing client-references to the *Frame*-object are adjusted. Clients have no way to access the attributes of the *Frame*-object directly, since they do not have a reference to that object.

The decorator-pattern is very powerfull. A slightly modified application of this pattern allows to extend a class in several directions (orthogonal). Let us look at another example:

In Chap. 6 we talked about an abstract *Stream*-class, which has several extensions like *Terminal, File* or *Network*. This presents an extension in *one* direction namely the output media. If we now want to have a *second* direction, which allows for different modes of encryption (e.g. RSA-, DES-encryption etc.) then we have a second (orthogonal) dimension. As can be seen in Fig. 9.14, every output-media may be combined with any encryption method. If this combination was to be implemented by subclassing, for every crossing-point on the grid we would need a separate subclass of Stream (i.e., *RSATerminal, RSAFile, RSANetwork* etc). The extensibility would be quite difficult. Introducing a new output medium, it would be necessary to combine it with every possible encryption method, which would yield a large number of new subclasses.

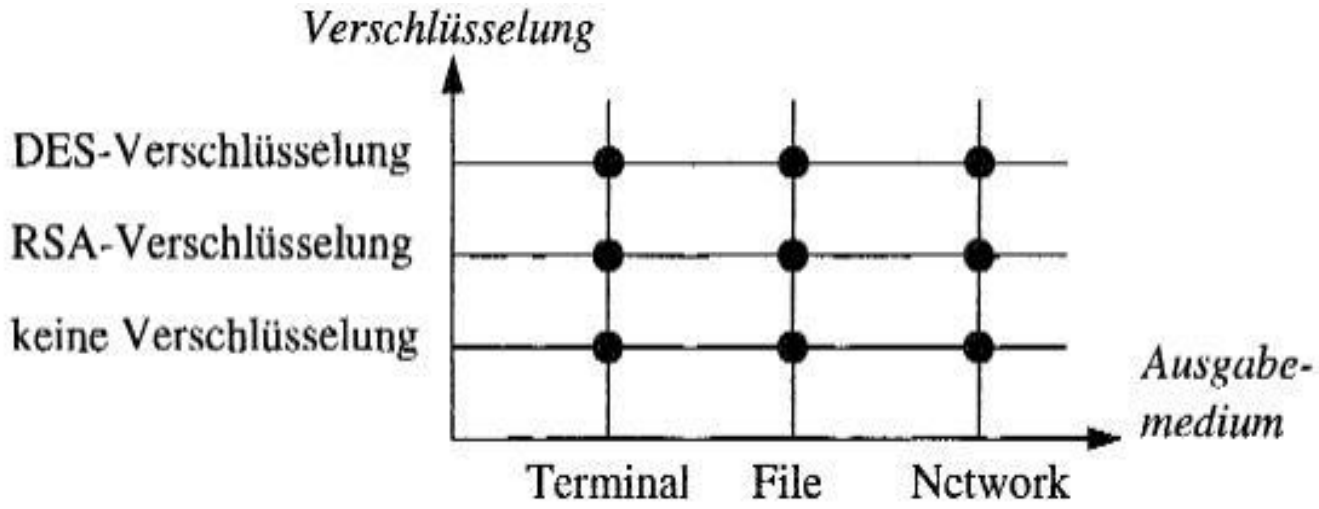


Fig. 9.14 Extension of a *Stream*-class for different media an encryption methods

The explosion of the number of classes can be avoided with the help of the decorator-pattern. The encryption method is put as decorator in front of the *Stream*-class. In Fig. 9.15 *Encoder*-takes the role of the decorator.

To combine a DES-encryption with a *File*, the *DESEncoder*-object is put in front of the *File*-object. If a *Write*-message is sent to the *DESEncoder*, the character *ch* is encrypted and afterwards with *stream.Write(ch)* forwarded to the *File*-object. It is possible to add new media (e.g. *MemoryFile*) or new encryption methods and nevertheless you can combine every output medium with every encryption method.

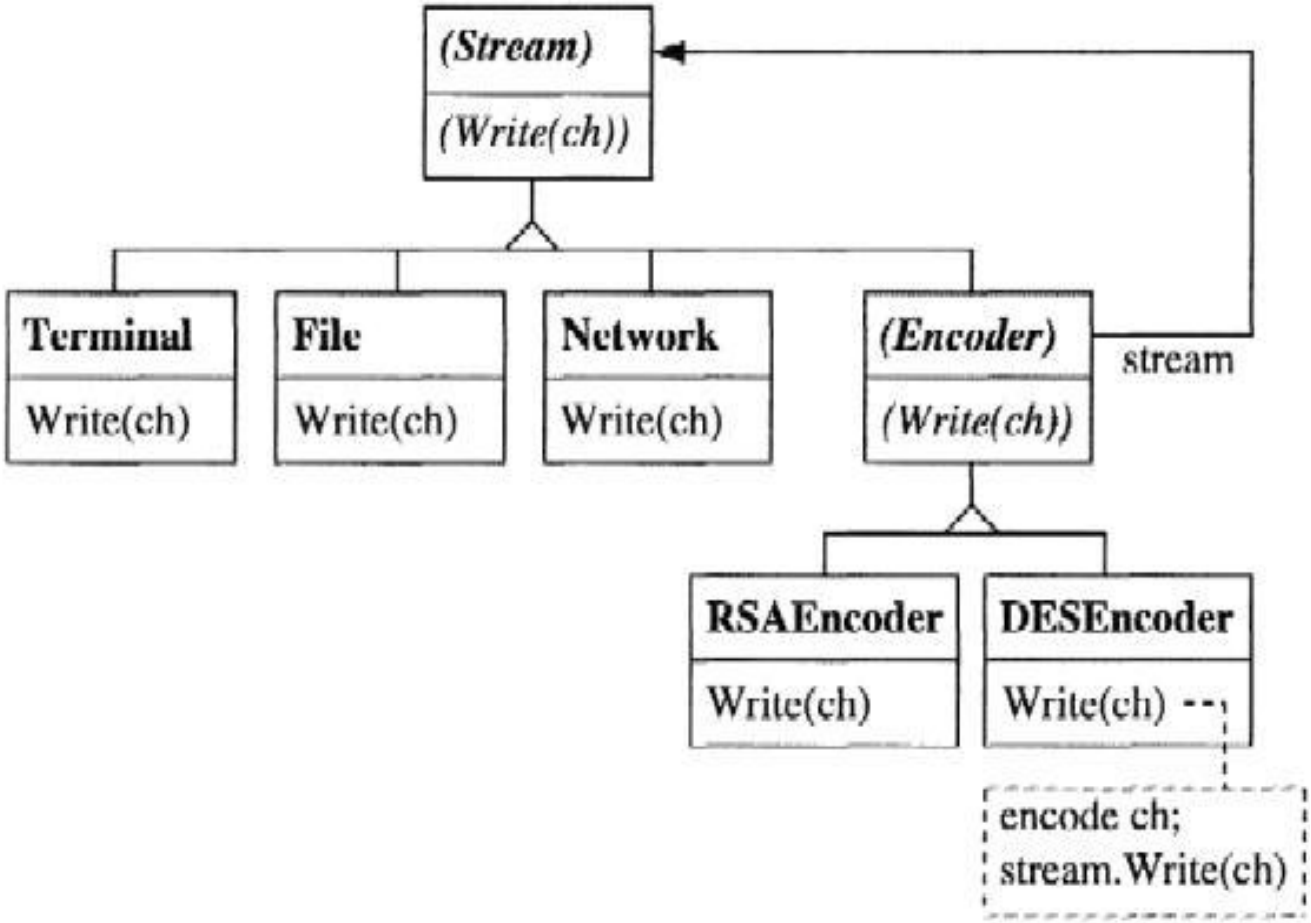


Fig. 9.15 Decorator pattern for extending *Stream* in two dimensions

9.3.5 Twin

Some languages like C++ or Eiffel offer multiple inheritance. As seen in Chapter 5, this allows on the one hand for inheriting attributes and methods from more than one base-class and - this might even be more important - make a subclass compatible with more than one base-class. On the other hand multiple inheritance also poses problems, like conflicting names or complex class-hierarchies. Therefore it is often tried to avoid multiple inheritance and to rely only on single inheritance. Languages like Oberon-2 and Component Pascal do not support multiple inheritance. Therefore we have to live with single inheritance. In this case, the *Twin*-pattern can be of some help.

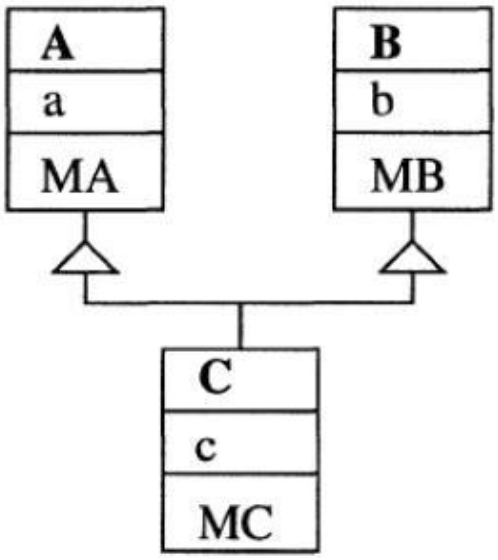


Fig. 9.16 Multiple inheritance

Fig. 9.16 shows the basic structure of multiple inheritance. A class *C* is derived from two classes *A* and *B* and inherits the attributes *a* and *b* together with the methods *MA* and *MB. C* is compatible with both *A* and *B. C* objects can be linked into a list of *A* objects and/or a list of *B* objects.

The twin-pattern is based on the idea to divide the class *C* into two twin-classes *CA* and *CB*, which are mutually linked with pointers (Fig. 9.17). *CA* is derived from *A* and *CB* is derived from *B*, for which single inheritance is sufficient. Attributes and methods of *C* are attached to one of the two twin-classes, e.g. *CA*.

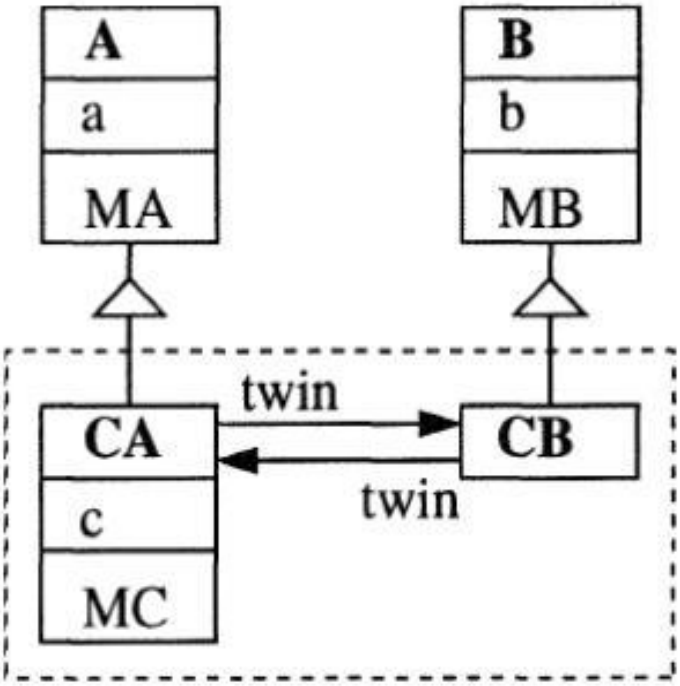


Fig. 9.17 Twin-classes *CA* and *CB*

Instead of the *C*-object a pair of a *CA*- and a *CB*-object is created. The *CA*-object can be inserted in a list of *A*-objects, the *CB*-object can be inserted in a list of *B*-objects. Therefore the twin-class is compatible with both *A* and *B* similar to the situation with multiple inheritance. If you want to override *MA*, you can do so in *CA*, if you want override *MB*, you can do so in *CB.*

Inherited attributes and methods can be accessed in the following way:

```
ca.a
ca.twin.b
ca.MA
ca.twin.MB
```

To access the components of *B*, you have to accept an indirection. This is the price to pay to avoid multiple inheritance, but it does not cost too much in most cases. It is possible to avoid the one level of indirection for calling *MB*, if you declare *MB* as a method of *CA*, which forwards the message to *CB* (Fig. 9.18).

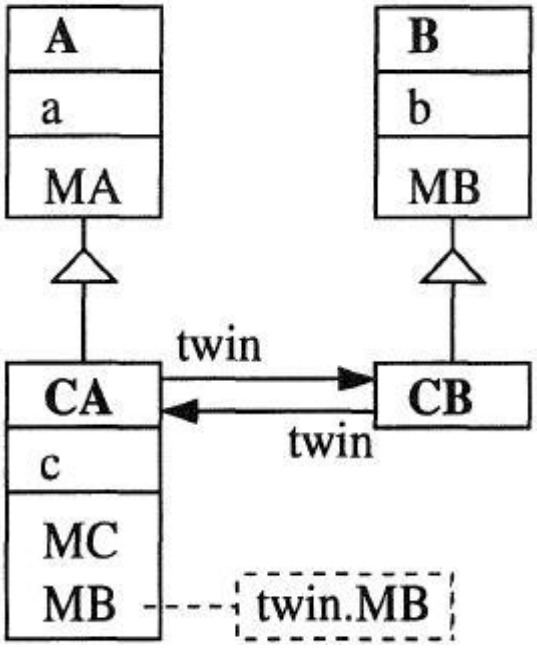


Fig. 9.18 Twin-pattern

The following example shows the twin-pattern again in a concrete application. Let us assume, a computer game has artifacts like balls and paddles, which are derived from a common class *Item*. Balls are *active* artifacts, which are in continous movement. Such artifacts are derived from a base-class *Process;* this yields a class-diagram shown in Fig. 9.19, which shows the multiple inheritance implementation. Balls can be members of a list of artifacts and of a list of processes.

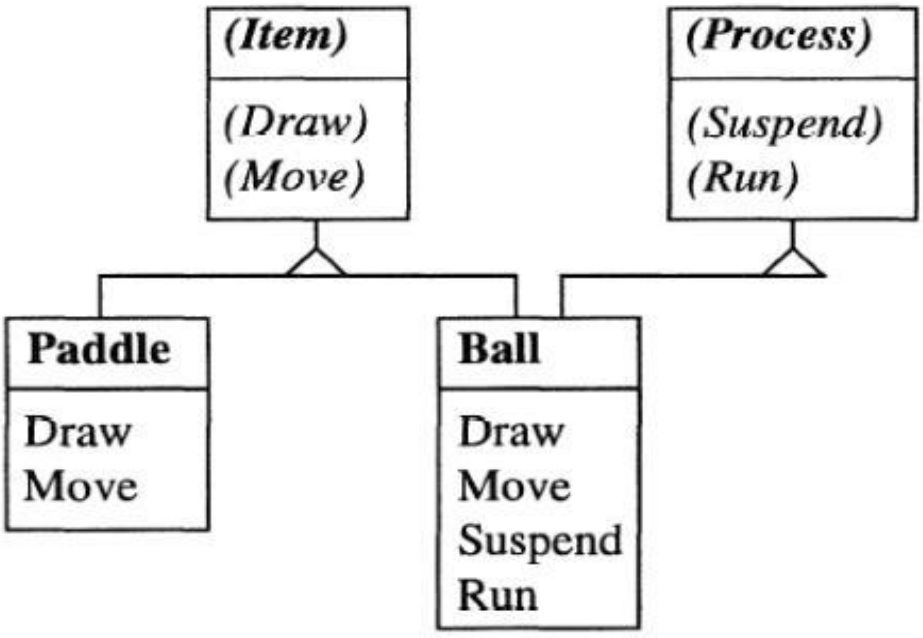


Fig. 9.19 Ball-game with multiple inheritance

Objects derived from *Process* receive a *Run*-message several times a second. A ball will move in reaction to this message. With these many *Run*-messages an impression of a continous movement of the ball on the screen is elicited. If the user hits any key, all processes get a *Suspend*-message. A ball reacts by freezing and changing color.

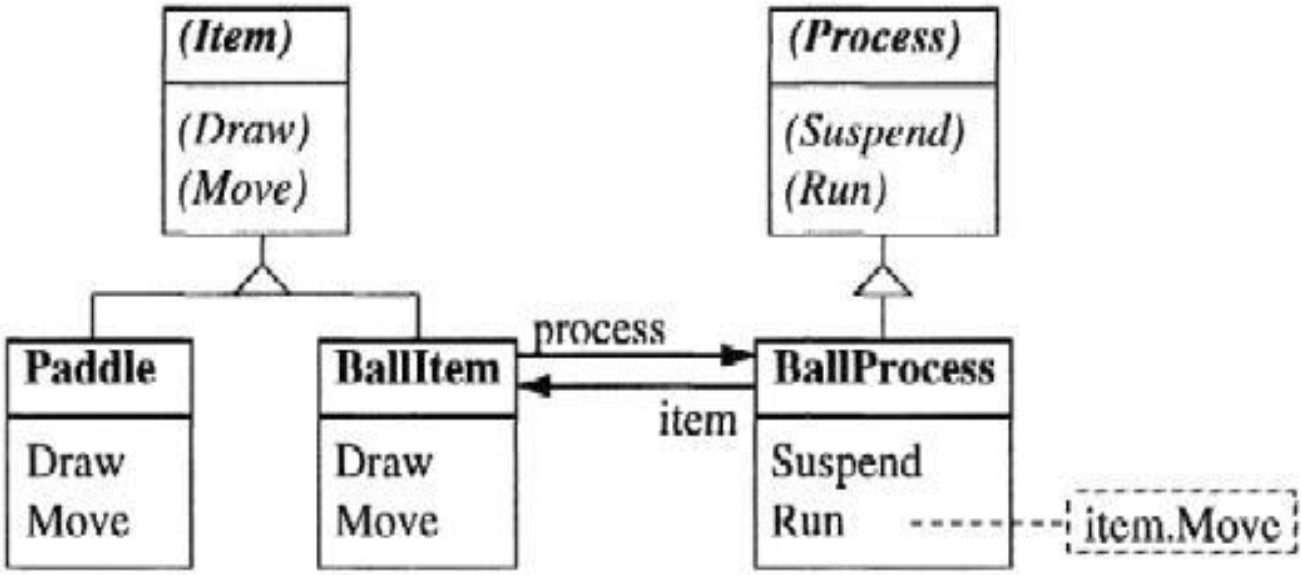


Fig. 9.20 Separation of class *Ball* into a twin-class pair *BallItem* and *BallProcess*

Let us now model this situation with the help of the Twin-pattern. The class *Ball* is separated into two classes *BallItem* and *BallProcess* (Fig. 9.20). *BallItem*-objects are linked into the list of artifacts, *BallProcess*-objects are linked into the list of active processes. If the *BallProcess*-object receives a *Run*-message, it acesses its twin-object *item* and moves it on a bit. If the complete board is redrawn, all objects in the list of artifacts (including the *BallItems*) receive a *Draw*-message.

In this example, we were able to go without multiple inheritance and we fullfilled all requirements of compatibility between artifacts and processes.

9.4 Behavioral Patterns

The third category of design patterns covers the so-called behavioral patterns. These are several methods for solving problems around objects. We will address the following patterns here:

- Message object — Looking at a message as an object
- Iterator — Iterating over a set of objects
- Observer — Reacting on a state-change
- Template — Algorithm with intervention handles
- Clone — Cloning objects
- Persistence — Input and output of objects
- Extension — System extension at runtime

9.4.1 Message Object

Methods are only *one* possibility to handle messages. Another way is to take the term "send a message" literally: In this case a message is a *data-packet* (a *message-object*), which is sent over to another object for handling. For doing that, we need different kinds of message-objects and a method, which interprets these message-objects.

Let us return to our example with figures, rectangles and circles. Figures can receive the messages *Draw, Store* or *Move*. If we implement these messages as objects, the following structure results:

```
TYPE
   Message* = ABSTRACT RECORD END (* base type of all messages *)

   DrawMsg*  = RECORD (Messge) END;
   StoreMsg* = RECORD (Messge) rider: OS.Rider END;
   MoveMsg*  = RECORD (Messge) dx, dy: INTEGER END;
```

The concrete messages are extensions of the abstract class *Message* and contain the actual parameters as attributes (record-fields). Records of this type can be handed over to a so-called *message-handler*, which is, as shown in Fig. 9.21 a method of *Figure*:

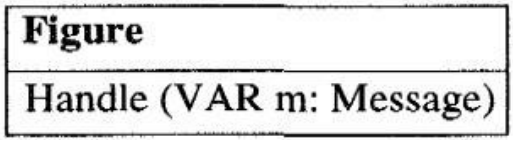


Fig. 9.21 Class *Figure* with message-handler

The message-handler *Handle* analyzes the incoming message-object according to its dynamic type and reacts. Every *Figure-class* overrides it accordingly. For the class *Rectangle* this is shown in Fig. 9.22:

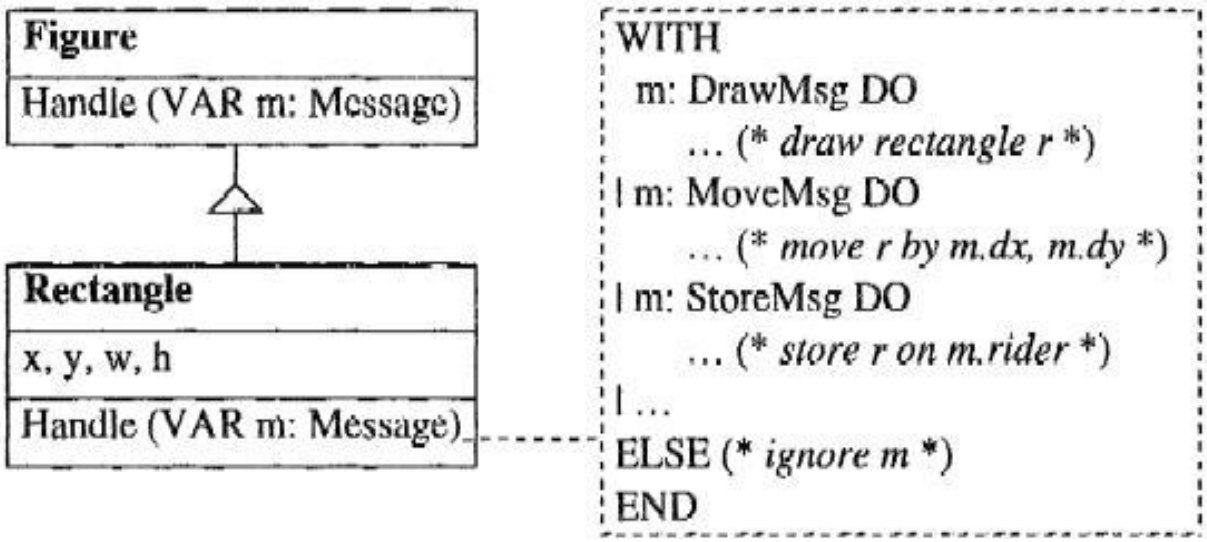


Fig. 9.22 Implementation of the message-handler in class *Rectangle*

For handling the message *m* a WITH-statement with variants is used, which can be interpreted in the following way: If *m* is of the dynamic type *DrawMsg,* the statement sequence following the first DO-symbol is executed; if *m* is of the dynamic type *MoveMsg,* the statement sequence following the second DO-symbol is executed; if no variant matches, the ELSE-branch is executed. If the latter is missing, a runtime error is generated.

In this example *Handle* ignores unknown messages: The ELSE branch of the WITH statement is empty. But it would be possible to emit an error message or send unknown messages to a handler of the base class.

If you want to send a message to a figure, you set up an appropriate message-object and send it to the handler of the figure:

```
VAR
   t : Figure; move : MoveMsg;
...
move.dx := 10; move.dy := 20;
f.Handle(move);
```

Depending on the dynamic type *f,* the message interpreter handles the *move*-message differently.

Return values of a message are stored in the message-object. If it is necessary to calculate the area of a figure, we can send a message *getArea*. The message interpreter returns the area in the attribute *getArea.value*.

```
TYPE
   GetAreaMsg = RECORD (Message) value : INTEGER END;
VAR
   getArea : GetAreaMsg;
   area : INTEGER;

f.Handle(getArea);
area := getArea.value;
```

Object-oriented programming using message-objects is similar to the way *Smalltalk* handles messages. In Smalltalk a runtime handler interprets messages and takes care of calling the corresponding method. But the message handler in Smalltalk is built into the system;

in Oberon it must be implemented by the programmer. The event-model of *Java* also uses message-objects.

The whole Oberon-system was implemented using message-objects. Also the Oberon0-system, which will be presented in Chap. 12 uses message-objects for handling windows on the screen.

Message-objects have the following advantages over methods:

- Message-objects are *data-packets.* They can be stored and sent later.
- A message-object can be handed to a procedure, which distributes the message to several objects (which may be unknown to sender). This is called a *broadcast.* Broadcasts are easily realized with methods, with only one exception: the sender knows every receiver and takes care that every receiver gets the message.
- Sometimes it is easier for the sender of a message that he does not have to care, if the receiver understands the message. Assume a list of different figures, where only rectangles and circles understand the *Fill*-message, lines don't. It is easier (but also more expensive) to send a *Fill*-message to all objects and leave it up to the objects, if they want to react. Otherwise it would be necessary to check first if it is possible to send that message to this object. With methods this approach is impossible, since the compiler already checks if a corresponding method exists in the class of the receiver.
- Finally it is possible to implement the handler not as a method but as a procedure-variable. In this case the handler can be substituted during runtime and the behaviour of the object can vary dynamically.

Message-objects also have drawbacks:

- You can't see, by looking at a class, which messages you are allowed to send to it. Although you can guess it by the different

types of message records, it is not necessary that all message-record types are declared in the same module. To be sure, you have to look at the implementation of the message-handler.

- The message handler analyses the messages *at runtime* with a WITH-statement. The branches of this WITH statement are processed sequentially. This is generally slower than a method call, which is usually implemented by a direct access to a method table (see Appendix A.12.4).
- Sending a message with message-objects involves writing more code than a method call. First the input parameters have to be packed into the message record, then the message-handler is called, and finally the results have to be fetched from the record.

```
msg.inPar := ....;
obj.Handle(msg);
... := msg.outPar
```

- What has been seen as an advantage, can also be a drawback: the compiler cannot check if an object understands a particular message. The following code fragment is completely correct for the compiler:

```
TYPE
   NonsenseMsg = RECORD (Message) END;
VAR
   t : Figure;
   nonsense: NonsenseMsg;
   ...
   f.Handle(nonsense);
   ...
```

During runtime *f* won't understand the message *nonsense*. The object will (hopefully) ignore the message. In the worst case the program will terminate with an error message. Such an error

may not occur for months and will be difficult to track down.

Message objects have advantages and disadvantages. Generally you should work with methods because this is more efficient, safer and more readable. In some situations (e.g. broadcasts) it can make sense to use the greater flexibility offered by message objects.

9.4.2 Iterator

You often want to apply a certain operation to a set of objects, but you don't know how to iterate through this set of objects due to the data-abstraction, which hides the concrete implementation (array, linear list, tree, etc.). An example of this is the class *Directory*, which manages a set of objects of the class *Element* (Fig. 9.23).

Fig. 9.23 Set of objects with unknown implementation

You don't know how *Dictionary* is implemented. What possibilities do you have for printing all elements of *Dictionary* ?

The simplistic approach is to provide a method *PrintAll* in *Dicitonary*, which prints all elements:

```
PROCEDURE (VAR d: Dictionary) PrintAll;
   VAR e: Element;
BEGIN
   e := d.firstElem;
   WHILE e # NIL DO e.Print; e := e.next END
END PrintAll;
```

The method *PrintAll* is local to *Dictionary* and therefore has access to its implementation. But this solution is unsatisfactory. For any other operation you need yet another method, e.g. *StoreAll* to store all

elements or *SelectAll* to select all elements for which a key matches a certain criterion. Additionally, these methods cannot use operations, which are defined in subclasses of *Element*.

Another solution is to declare an *Iterator*-class in the same module as *Dictionary*, as shown in Fig. 9.24

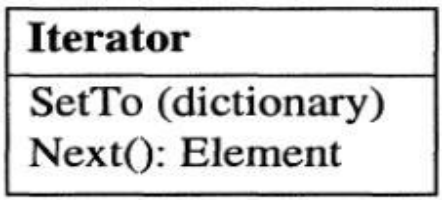


Fig. 9.24 *Iterator*-class

An *Iterator* is an object which can be moved through a data-structure. *SetTo* sets the iterator to the start of the data-structure and *Next* yields the next element of it. With the aid of an iterator all elements of *Dictionary* can be visited sequentially and arbitrary operations can be applied.

```
Iterator.SetTo(dictionary);
elem := Iterator.Next();
WHILE elem # NIL DO
    elem.Print;
    elem := Iterator.Next()
END
```

This second solution is universal, but it requires that the code for iterating be implemented in each client. A problem results when the data-structure is e.g., a tree, which is efficiently visited recursively. In this case it is difficult to implement *Next* efficiently.

The result-type of *Next* is *Element*. The actual type of the result might be an extension of *Element* (e.g. *MyElement*). By applying a type-guard, it is possible to send *MyElement*-messages also to the result of *Next*, which were not foreseen in *Element*:

```
Iterator.SetTo(dictionary);
elem := Iterator.Next();
WHILE elem # NIL DO
   IF elem IS MyElement THEN
      elem(MyElement).Store(rider)
   END;
   elem := Iterator.Next()
END
```

A third possibilty is to use message-objects. You call the dictionary with a message-object, which implements the operation to be applied to the elements. The message-object is then sent to all elements. Each element must have an interpreter which reacts to the message-object. But this solution appears to be too costly for simple applications like printing all elements.

Finally it is possible to provide in *Dictionary* a universal method *ForAll* with a procedure parameter. This procedure will be called for all elements:

```
PROCEDURE (VAR d: Dictionary) ForAll (P: PROCEDURE(e: Element);
   VAR e : Element;
   BEGIN
      e := d.firstElem;
      WHILE e # NIL DO P(e); e := e.next END
END ForAll;
```

A call of this method may look like:

```
dictionary.ForAll(Print);
dictionary.ForAll(Store);
```

where *Print* and *Store* are procedures of the client:

```
PROCEDURE Print (e: Element);
   BEGIN
      e.Print
END Print;

PROCEDURE Store (e: Element);
   BEGIN
      e(MyElement).Store(rider)
END Print;
```

In this way it is possible to apply almost any operation to the elements of the set.

This last solution is the most simple and readable solution for Oberon-2 and Component Pascal. Some other languages (e.g. Sather) have special iterator constructs or so-called block-objects (e.g. Smalltalk), which allow iterators to be implemented even more simply.

9.4.3 Observer

An observer is an object which is interested in the state of another object. It registers itself at the other object as an observer and it will be informed when the state of the observed object changes.

The observer pattern occurs frequently with graphical user interfaces. Fig. 9.25 shows an example of a measurement-object, which represents a number between 0 and 100. Two graphical objects - a slider and a meter - are registered as observers of the value of the measurement. If this value changes, the slider and the meter are notified and refresh their graphical representation accordingly.

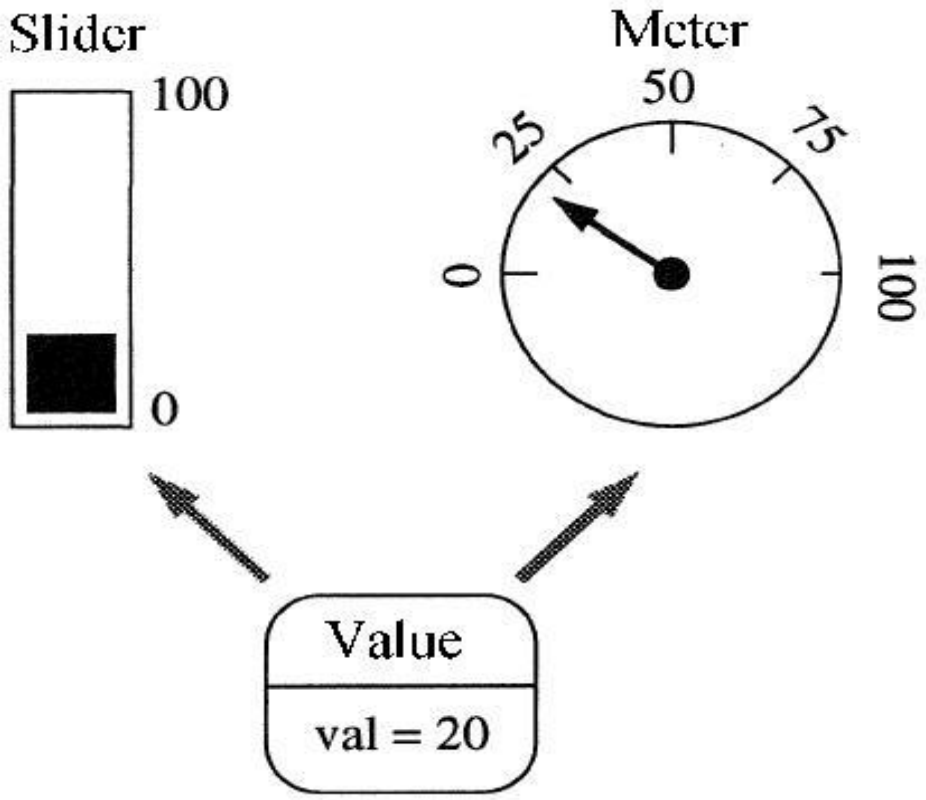


Fig. 9.25 Slider and meter as observer of a measurement

The observer-pattern serves another purpose. It guarantees the *consistency of all observers* of an object. Since the the observers are immediately notified of any state change of the object, they know at any time its actual state and are therefore consistent among each other.

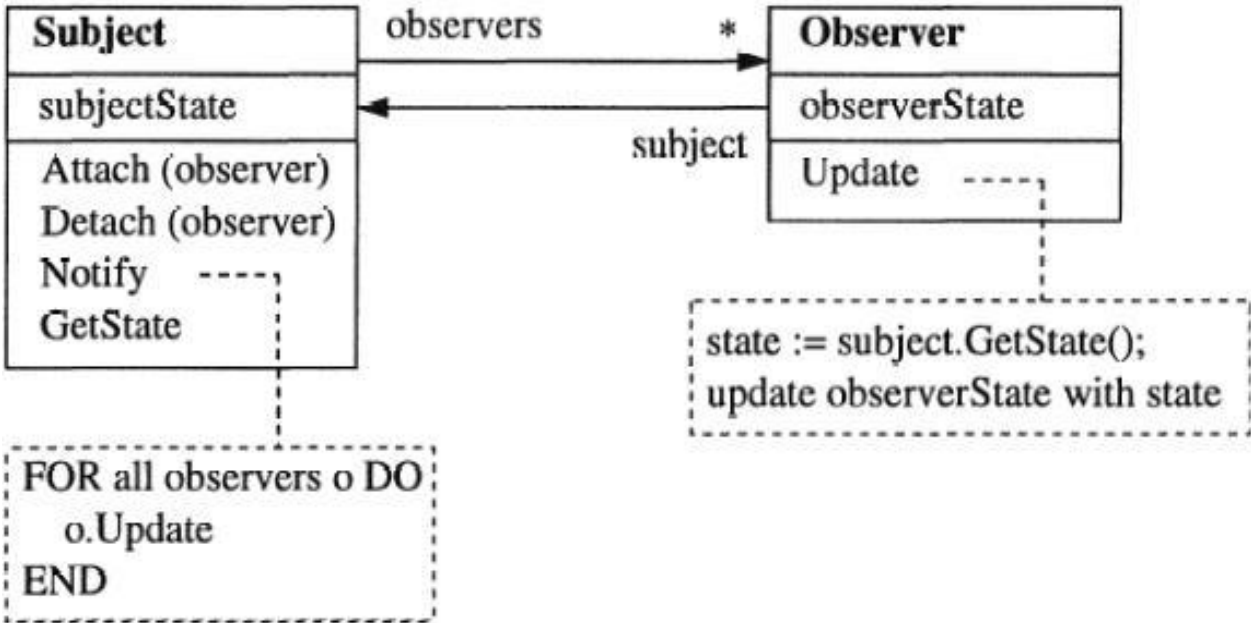


Fig. 9.26 Observer pattern

The class-diagram in Fig. 9.26 shows the observer pattern together with its class and its relations. *Subject* (the observed object) sends itself a *Notify*, when its state changes. The *Notify*-method in turn sends all registered observers an *Update*. The observers fetch the current state from *Subject* by calling *GetState* and refresh their own state.

Please note that observers can dynamically register and unregister themselves (with *Attach* and *Detach*). The relation between *Subject* and its observers is therefore only a temporary one and can be changed during runtime. It may be the case that no observers at all are registered. In this case *Notify* will be a no-op and nobody will be notified of a contingent change of state.

If an observer gets notfied of a state-change, he needs information about *what* has changed. There are two possible solutions: the new state can be handed over by *Update* as a parameter (*Push-model*), or the observer only gets information about which aspect of the state changed. In the latter case, the observer has to retrieve the information relevant for him by calling *GetState* or a similar method (*Pull-model*). Pushing is obviously better suited for simple state changes, and pulling for complex ones.

Fig. 9.27 shows the interactions between a subject and its observers with the help of an *interaction-diagram*. The vertical lines represent objects, the horizontal arrows represent messages. The bars on the object lines represent the lifetime of the methods called. Interaction diagrams like this one are well suited to describe dynamic properties of programs.

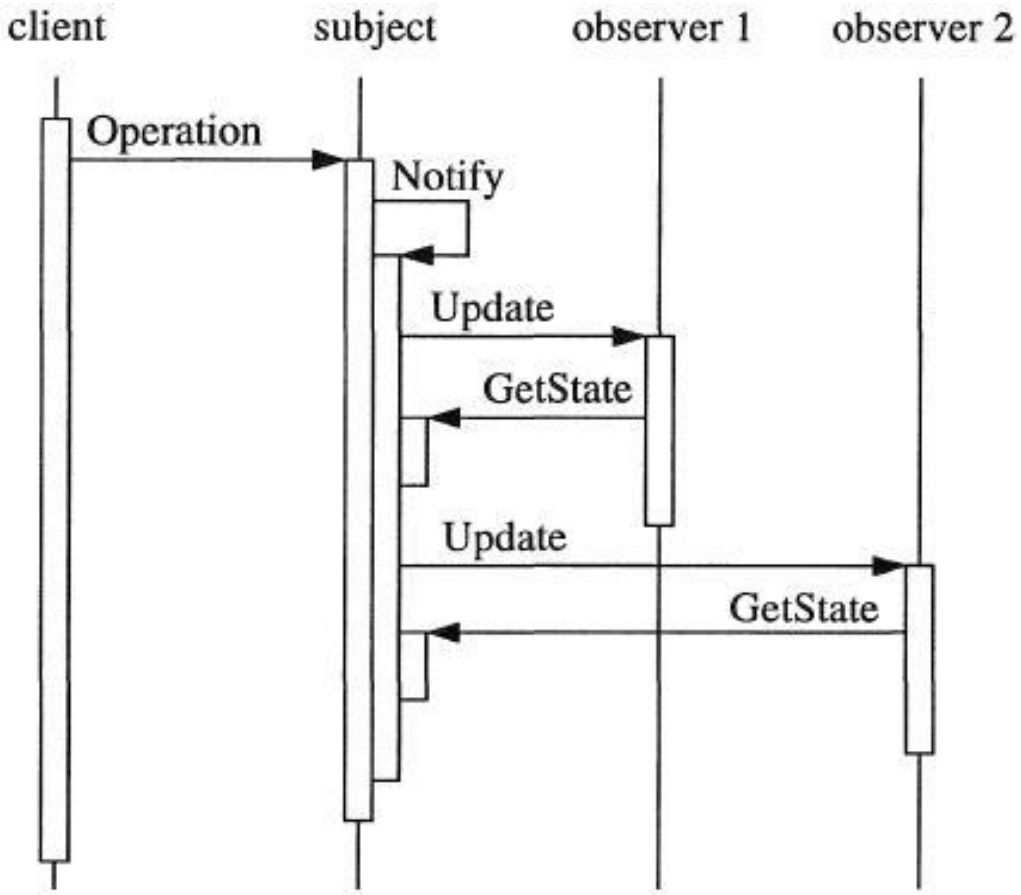


Fig. 9.27 Interactions in the observer pattern

9.4.4 Template Method

A *template method* defines an algorithm by a sequence of calls to abstract methods. It thereby fixes a series of steps, but the implementation of these steps is left open; the latter is done by implementing the abstract methods in subclasses.

Let us consider as an example a class *Frame* for windows on a screen. When a certain region of the window should be redrawn, it is necessary to remove a possible selection, to define a clipping area and then to redraw the contents of the window. These three steps are not dependent on the window being a text or a graphic window. They can be merged into a template method *Restore*, as shown in Fig. 9.28.

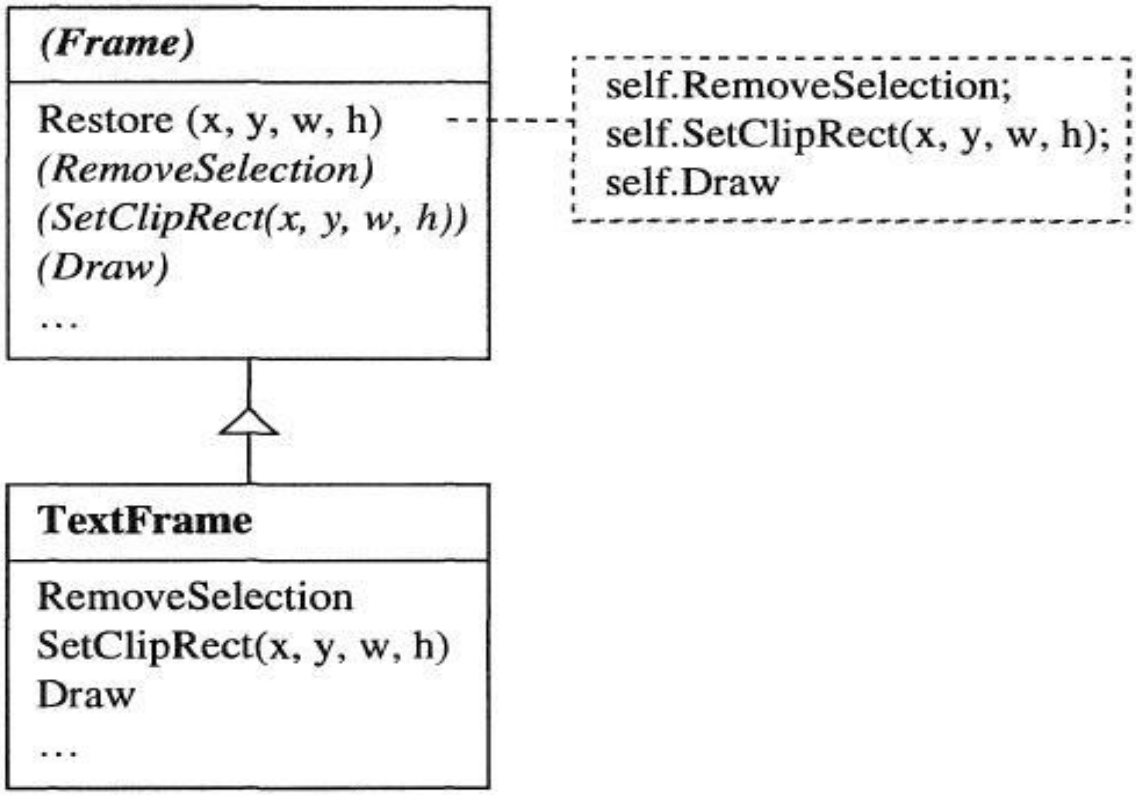


Fig. 9.28 Template method *Restore*

Needless to say, the methods called by *Restore* can not yet be implemented in the abstract class *Frame*, since they will be different for text frames and graphic frames. An example of the implementation might be in the subclass *TextFrame.* Nevertheless *Restore* already has fixed the correct sequence of calling them. It is now possible to send a *Restore* message to a text-frame and it is not necessary to care about the individual sub-steps. Additionally, the template method ensures that all *Frame*-classes undergo these sub-steps in the same sequence.

It is not necessary that the sub-steps are purely abstract. Some of them can implement a particular algorithm and only call empty methods at distinct places. The programmer has the opportunity to override these (empty) methods and to gear into the course of the template method. These empty methods are called *hooks*, since they provide a chance to hook own code into an existing algorithm.

Let us look again at an example: a graphic window has a method *TrackMouse,* which tracks mouse movements and draws the mouse pointer. If you want to restrict the movement of the mouse-pointer to a

fixed grid, you can call an empty method *Constrain* by default. But it is possible to override this empty method *Constrain* by sub-classing in such a way that the provided mouse-coordinates are constrained to the nearest grid-point (Fig. 9.29).

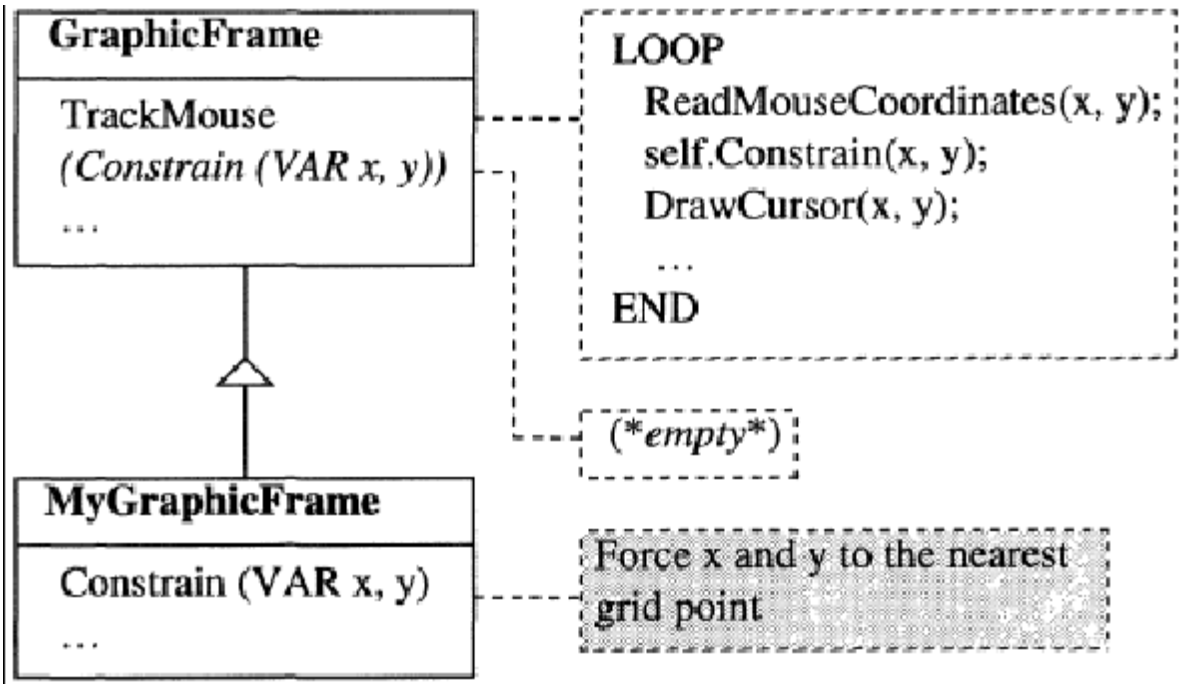


Fig. 9.29 *Constrain* as a hook in the method *TrackMouse*

Providing as many hooks as possible is a usual technique to make an algorithm as flexible as possible. If you provide too many hooks, the efficiency of the algorithm may suffer and the interface of the template method may become unmanageable.

9.4.5 Clone

It appears to be a trivial task to copy or clone objects. But if you don't know the dynamic type of the copy source, the task is not that trivial. How to proceed?

An obvious solution is to send the object a copy message. With the help of dynamic binding, the *Copy*-method of the runtime-type of the object is called. The *Copy*-method allocates a new object and initializes the attributes with the values of the original. This apparently

simple solution has its own pitfalls, as shown in Fig. 9.30.

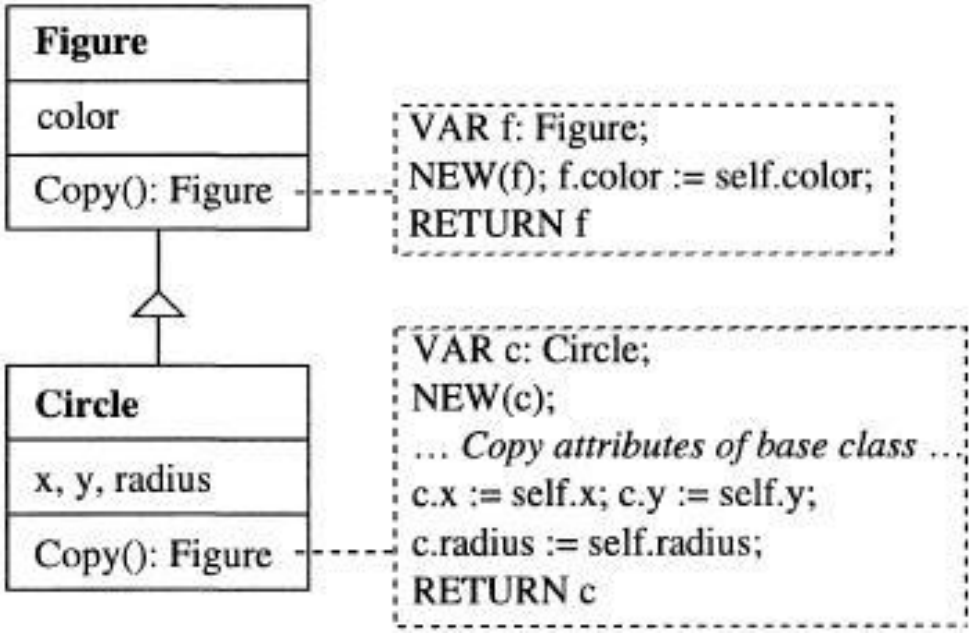


Fig. 9.30 Problems when copying objects

The *Copy*-method of *Circle* has to copy not only its own attributes, but also the attributes of the base-class *Figure*. This is only possible for the exported attributes of *Figure*. It is not possible to use the *Copy*-method of *Figure*, since this method will create a new object instead of copying only the base-attributes to the new object *c.*

The solution here is a simple trick: hand the new object over to *Copy* in an additional VAR parameter. If this parameter is NIL, the *Copy*-method knows that a new object has to be allocated before copying the attributes, otherwise the input object is recycled. This behaviour is illustrated in Fig 9.31.

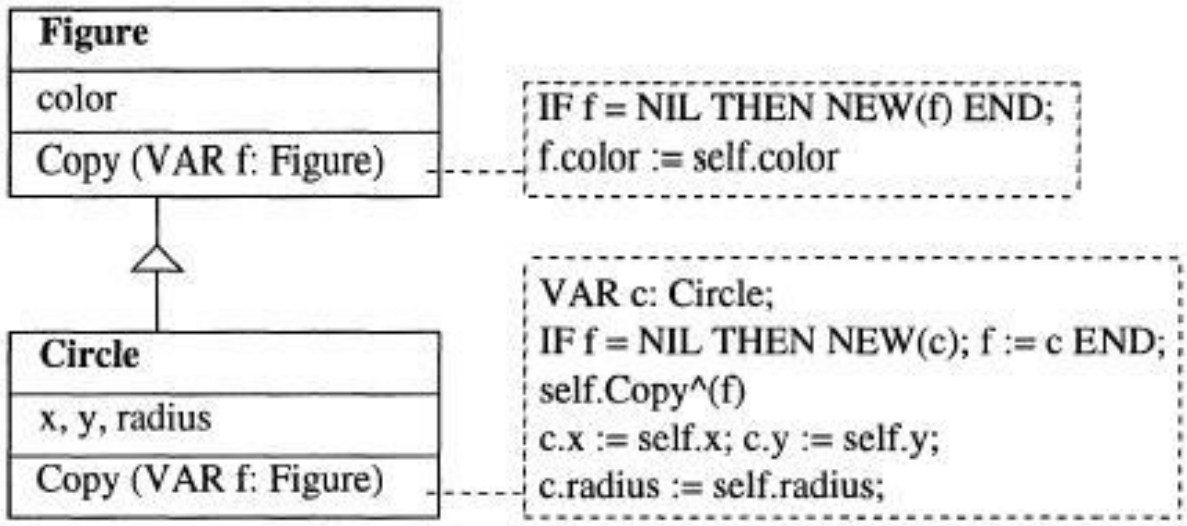


Fig. 9.31 Copying of objects

To instantiate a copy *f* of an object *g* you write:

```
f := NIL; g.Copy(f)
```

If *g* has the dynamic type *Circle* the *Copy*-method of *Circle* is called. Since *f* has a value of NIL, a new *Circle*-object is allocated. When calling the base-class *Copy*-method, the value is different from NIL. Therefore no new object is allocated, and only the *color*-attributeis copied. Example code

It is slightly awkward to be forced to set the parameter of the *Copy*-method to NIL. It is possible to avoid this necessity if the runtime system provides operations for working with types and for the creation of objects of certain types. The Oberon-System provides such operations in module *Types* (see App. B). *BlackBox* offers them in module *Kernel*. The relevant parts of these modules are:

ð BlackBox:

```
DEFINITION Kernel;

   TYPE
      Name = ARRAY 256 OF SHORTCHAR;
      Module = POINTER TO RECORD
         ....
         name-: Name
      END;
      Type = POINTER TO RECORD
         mod-: Module;
         id-: INTEGER;
         base-: ARRAY 16 OF Type;
         fields-: Directory;
         ptroffs-: ARRAY 1000000 OF INTEGER
      END;

   PROCEDURE TypeOf(IN rec: ANYREC): Type;
   PROCEDURE NewObj(VAR obj: SYSTEM.PTR; t: Type);
   PROCEDURE ThisType(mod: Module;
                      name: ARRAY OF SHORTCHAR): Type;
   PROCEDURE GetTypeName(t: Type; VAR name: Name);
     ...
   END Kernel.                                          ï
```

Type is a *type-descriptor*, i.e., *Type* describes certain properties of a type like its name or the module in which it is declared. If *p* is a pointer to a record of type *T* the *TypeOf(p)* returns the type-descriptor of *T* (SYSTEM.PTR is a type, which is compatible with any pointer type). *NewObj(t, obj)* creates a new object *obj* of the type descibed by the type-descriptor *t*. In the Oberon System *This(m, name)* and in BlackBox *ThisType(m, name)* returns the type-descriptor of the type with name *name* declared in module *m*.

With the aid of the low-level module *Types* or *Kernel*, *Copy* can be implemented more elegantly:

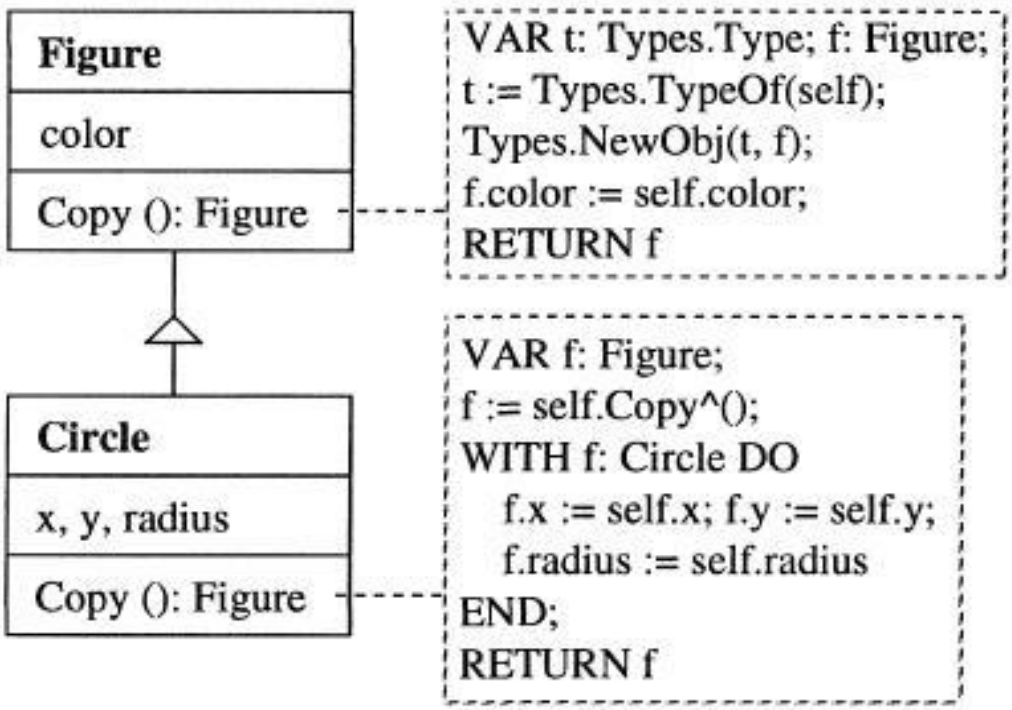


Fig. 9.32 Copying of objects with means of the runtime system (Types in the Oberon-System, in BlackBox: IMPORT Types := Kernel)

To create a new object *g* the following suffices:

```
f := g.Copy()
```

If *g* has the runtime type *Circle*, the *Copy*-method of circle is called, which in turn calls the *Copy*-method of the base-class. With the help of module *Types* a new object *f* with the same type as the receiver *g* is created, in our case therefore a *Circle*-object. After copying the attribute *color, f* is returned to the *Copy*-method of *Circle*. The remaining attributes are copied in *Circle*. The WITH-statement changes the static type of *f* to *Circle,* therefore the attributes *f.x, f.y,* and *f.radius* are accessible for the copy-operation. Example code

9.4.6 Persistence: Input/Output of Objects

In almost every program it is eventually necessary to write objects to a file and read them again later. In this situation a similar problem arises as in the case of cloning objects: How is it possible to read and write

objetcs of unknown dynamic type?

Writing such objects is not really that difficult. Simply send an object a *Store*-message and trust that the object itself knows which attributes it has to write to a file. Reading objects is significantly more difficult: Before you can read an object, you have to create it. But do you know the type of the object which is to be created? You don't have any idea about the dynamic type of that object.

The solution is to store in the file not only the attributes of an object, but also its *type-name.* By using module *Types* (BlackBox: *Kernel*) it is possible to retrieve the name of any type from an object and in turn to create any object by its type-name. The following two procedures *StoreObj* and *LoadObj* write an arbitrary object including its type-name to a file and read it back:

ðBlackBox:

```
PROCEDURE (VAR r: OS.Rider) StoreObj* (x: Object);
   VAR type: Kernel.Type; name : Kernel.Name;
BEGIN
   IF x = NIL THEN r.Write(0X)
   ELSE type := Kernel.TypeOf(x); Kernel.GetTypeName(type, name );
      r.WriteString(type.name); r.WriteString(type.name); x.Store(r)
   END
END StoreObj;

PROCEDURE (VAR r: OS.Rider) LoadObj* (VAR x: Object);
   VAR name1, name2: ARRAY 32 OF CHAR; type: Types.Type;
BEGIN r.ReadString(name1);
   IF name1 = "" THEN x := NIL
   ELSE r.ReadString(name2);
      type := Kernel.ThisType(Kernel.ThisMod(name1), name2);
      Kernel.NewObj(x, type); x.Load(r)
   END
END LoadObj;
```

ï

Both methods assume that all objects under consideration are derived from a class *Object*, which support the messages *Load* and *Store*. The type *OS.Rider* represents a read/write-position in a file (see also

App. B.5). *StoreObj* fetches the type-descriptor of *obj* and writes the module name (*type.module.name*) and its type-name (Oberon System: *type.name;* BlackBox: *Kernel.GetTypeName)* to the file. *LoadObj* reads module- and type-name, fetches the module-descriptor *mod* and the type-descriptor *type* and generates a new object of this type with *NewObj*.

Whereas *StoreObj* could be implemented in principle also as a method of *Object*, this cannot be achieved with *LoadObj* because it is impossible to send a message to an object, which does not (yet) exist.

Fig. 9.33 shows how the *Load-* and *Store*-methods of a class *A* and its subclass *B* can be implemented in such a way that all attributes can be correctly read and written. Please note that *B* has an attribute *b*, which is itself an object. Therefore this attribute is output by *StoreObj* and read back with *LoadObj.* Reading back needs a type-guard since *LoadObj* returns a parameter of static type *Object.*

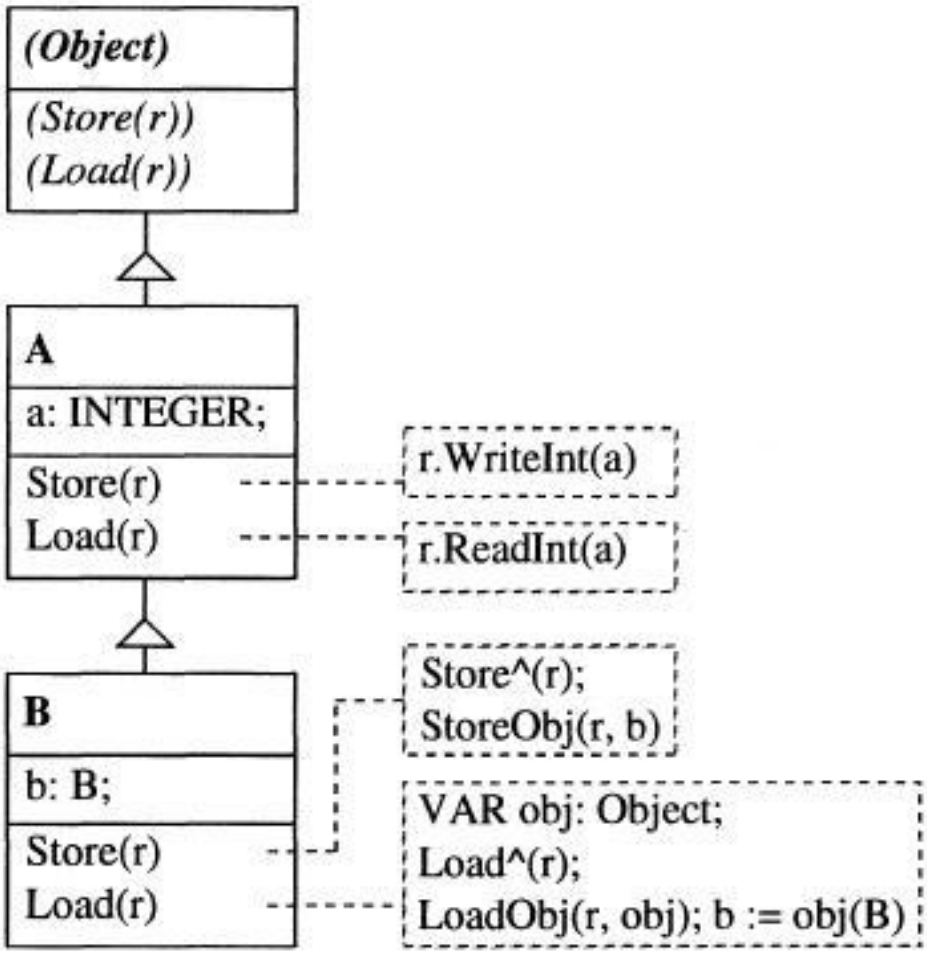


Fig. 9.33 Implementation of *Load*- and *Store*-methods

If the runtime system does not provide means to convert an object into its type-name and vice versa, the following workaround can be used: Each object must support a message *GetTypeName*, which returns the type-name of the object. Furthermore a table must be allocated, where for each used type a prototype is stored together with its type-name. If you need an object with a certain type-name, the table is searched and the corresponding prototype is cloned. Clearly this needs more effort on the side of the programmer than the solution sketched above.

Reading and writing objects is the basis of the implementation of persistent objects. An object is called *persistent* if it survives the program which has created it. Persistent objects are used in database-like applications. They are often interweaved with other objects to a graph-like web. Reading and writing such a web needs attention to ensure that each object is written only once, even if there are several pointers to that object. Techniques for such applications can be found in the standard literature on graph-algorithms.

9.4.7 Runtime Extension of a System

In Chap. 8.3 we have seen that a graphic-editor can be extended at runtime to support new objects (rectangles, circles, lines) which are unknown at the time of the initial implementation of the editor. Here we want to take a look at how this can be achieved in Oberon without the requirement to unload, re-link, and re-load the program which is to be extended.

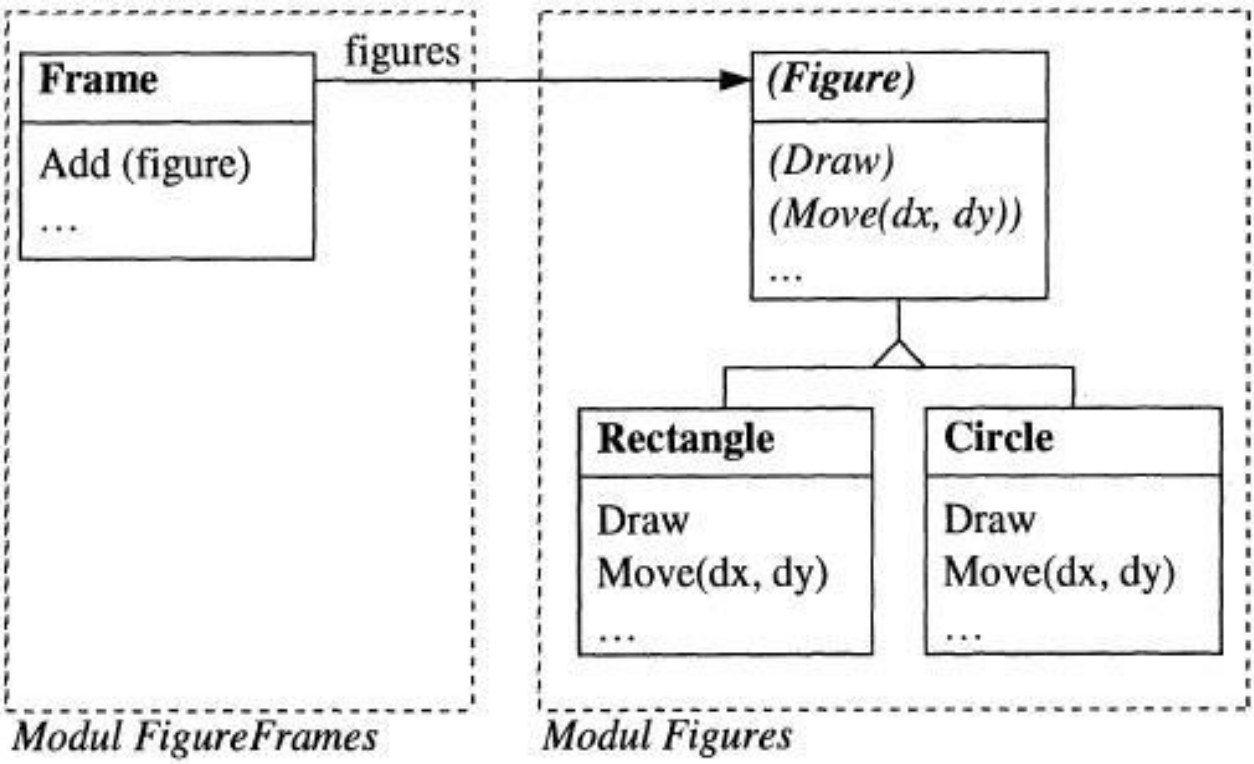


Fig. 9.34 Figure-family in a graphic-editor

Let us recall the example of Chap. 8.3: the graphic-editor works with a *Figure*-family consisting of rectangles, circles or other figures (Fig. 9.34). We assume that all classes of this family are declared in a module *Figures*. One of the modules of the editor is *FigureFrames* with a class *Frame*, which is responsible for managing the figures and displaying them on the screen. There is a method *Add* for adding new figures.

That is the core of the editor. During implementation it is not (yet) necessary to know, which kind of figures will come later. The editor can handle any sub-classes of *Figure*.

If we now want to extend the editor with ellipses, the following needs to be done:

1. Define a class *Ellipse* as sub-class of *Figure* and override the inherited methods (Fig. 9.35)
2. Implement a command *New*, which creates an *Ellipse*-object and inserts it into the list of the other figures in the actual frame.

```
PROCEDURE New;
   VAR e: Ellipse;
   BEGIN
      NEW(e);
      ... (* fille e.x, e.y, e.a, and e.b *)
      FigureFrames.currentFrame.Add(e)
END New;
```

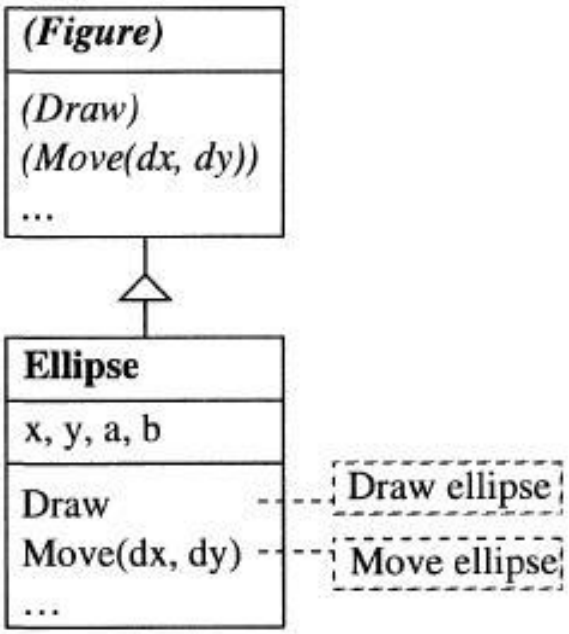


Fig. 9.35 Deriving a new subclass *Ellipse* from *Figure*

The class *Ellipse* and the command *New* are wrapped in a new module *Ellipses*. It is not necessary to touch the existing modules of the editor. To install a new *Ellipse*-object in an editor window, call the command *New*. The following happens:

1. If module *Ellipses* is not yet loaded, it will be loaded now and added to the editor, thereby extending the editor at runtime with a new module.
2. The command *New* is executed. It creates a new *Ellipse*-object and inserts it into the figure-list of the actual frame.
3. The frame sends the newly inserted figure (where the editor does not have know the new figure's type) a *Draw*-message, which causes the ellipse to be drawn.

It should be noted that the module *Ellipses* will be loaded on demand and bound to an already running editor-core. Neither *Figures* nor *FigureFrames* know (i.e., import) *Ellipses.* Therefore both can be compiled and used long before *Ellipses* is created. Conversely *Ellipses* imports *Figures* and *FigureFrames*. *Ellipses* builds up on these modules and extends them.

The editor-core can use the unknown module *Ellipses* as a result of the dynamic binding. The editor-core sees the *Ellipse*-object as an incarnation of the abstract class *Figure* and communicates with it via messages, which yield to calls to methods from module *Ellipses,* which is higher up in the import hierarchy. Therefore these calls are called *up-calls.*

The chance to extend a system during runtime without unloading, re-compiling, and re-linking is one of the main advantages of object-oriented programming and build its strength. The Oberon-System (and the BlackBox component framework), with its commands and the dynamic loading of modules, offer the preconditions for this extensibility.