Programming Concepts and Design Patterns

CHAPTER 7

Computer programmers develop complex programs that contain many algorithms. This is a very challenging task because you cannot see easily that an algorithm is correct or that its program representation is functioning as expected. The more complex a program the more likely is that it contains errors.

Huge programs have been built that were not any more manageable. When maintenance or correction were made, new errors have been introduced. The situation was also known as "software crisis".

Software developers needed guidelines and rules how to make good programs. In the beginning of computer programming until the 60^{th} of the last century there were no such rules for "good" programming. If a novice programmer has written some small programs he might be tempted to believe that the construction of a large program is similar, only more time and men power are required. This is a big mistake because the complexity of such a project does not increase linearly with its size but with square order. The nonlinear growth becomes plausible if you consider not only the number of code lines but also the number of references or dependencies between code lines or statements. Furthermore, the management effort of a programming project depends on the communication effort that is growing with square order of persons involved.

Program design techniques are vital for a predictable, manageable and reliable software development. Many methods and techniques have been developed to support this goal.

This chapter is dedicated to the concepts and design techniques of programming.

CHAPTER OBJECTIVES

- To know how to "divide and conquer" a complex task
- To appreciate the benefits of modularization and information hiding

- To introduce repetition concepts
- To understand how to apply design patterns

Program design is about concepts, rules, methods, and techniques to structure a software system. The shining example was engineering where a large wealth of proven engineering techniques exist for the construction of large buildings, machines, and electronic devices. These techniques support the development of products in a predictable manner regarding costs, resources, time, and quality.

Software Engineering is not mature enough for a similar reputation. Beside that, the situation is different: The artifacts produced in programming are different to that of engineering in some important aspects:

Software	Hardware
abstract	real
no tolerance	tolerance
lack of metrics	physical metrics

Software is *abstract*, only its textual representation (the source code) is visible. The textual representation only reflects a static view of the program, it does not reveal the dynamics and its behavior during execution. This makes it difficult to develop a program, test its functionality, and evaluate its quality. Software usually is not allowed any *tolerance*. Multiplying two numbers is always expected to yield the same result. Logical behavior has to be independent of time, space, and hardware that execute the instructions.

This is quite different to mechanical engineering where each part, even a simple screw have tolerances in length, thread, and hardness. In engineering we have clear quality criteria based on physical *metrics*. For software products, however, it is hard to objectively measure its quality due to the lack of adequate metrics. It is relatively easy to find a metric for the quality aspect *error* of a program. You can count the number of errors and even quantize its severity. But, for the quality aspect *ergonomic* it is much more difficult because some of the ergonomic criteria are competing, i.e. if you improve one criteria you worsen an other criteria at the same time. A similar problem causes the *security* aspect. Any metric should take in account all possible threats and vulnerabilities. But this is not known and in general not provable.

In spite of this difficulty there are many rules, concepts, and techniques for good software design and how to manage the complex process of program development.

Let's start with a rather old principle and insight how to cope with a task that is to difficult to handle in one piece.

7.1 Divide and Conquer

The structuring of a task into smaller tasks is one of the most powerful techniques to master complexity not only in programming. If you have a heavy

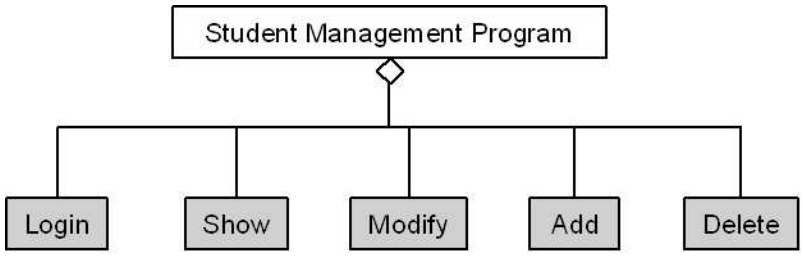

Figure 7.1 Breakdown of a program into functional units

load to carry you might want to partition it into smaller pieces so that you are able to carry them easily.

The **partitioning principle** is known under the Latin dictum "divide et impera" (divide and conquer) which is attributed to the Roman emperor *Julius Caesar*. Applied to ancient politics it meant to pit Roman's enemies against each other.

Applied to programming this principle means that a large and difficult program could be made easier to write by dividing the task into smaller pieces. Later in this chapter we will call these pieces procedure, function, method, or module.

To make the partitioning principle applicable we have to assume that a task can be partitioned and that the subtasks can be easily combined to get the complete task done.

Let's say we want to write a program that manages the student records. The program needs to administer student records and only authorized staff should be allowed to use it. We could partition the task into the following parts:

- login
- add (a student)
- modify
- delete
- show

Figure 7.1 visualizes this division in Unified Modeling Language (UML). The diamond symbolizes in the UML that the Student Management Program is an aggregate, i.e is build of the shaded parts that contain the functionality.

If we divide a program into n parts then we can compute the possible interconnections between them by:

$$(n-1)+(n-2)+(n-3)+\ldots+2+1 = n(n-1)/2.$$

Partitioning "squares" the complexity for the collaboration of the parts, i.e. if we double the parts we essentially quadruple the possible connections between them. This is not acceptable to reduce the functional complexity on the costs for structural and interaction complexity. But, if we restrict the connections

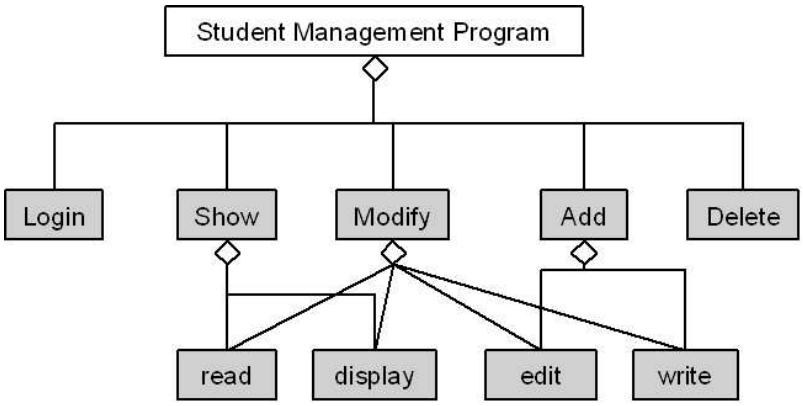

Figure 7.2 Breakdown of a program with reuse of parts

between the parts to form only a hierarchical or layered structure, the linking problem will grow only linearly or even less. So we can keep the interactions under control.

In our student administration example we built a hierarchical aggregate-component structure resulting in 5 connections.

In general, if *n* parts are directly connected to its aggregate, we obviously have *n* connections. This is a linear growth only compared to the quadratic growth in the general case above.

In our example the *modify* part could be further broken down to the following subtasks:

- read (data from file)
- display (data on the screen)
- edit (data on the screen)
- write (data to file)

We notice that *show* could be build of *read* and *display*, or *add* could be made of *edit* and *write*. So the same functions may be used at different places. The puzzle is put together in figure 7.2.

Stepwise refinement leads to smaller units that have a more general functionality. This apparently promotes the *reuse* of parts. We will take a closer look at reuse of program parts (modules) in the next section (7.1.1) of this chapter.

An other possibility of multiple use is repetition. As example take the function *read* and further refine it into positioning at the beginning of the record and then repeatedly reading character by character until the end of the record.

Most algorithms contain operations that are executed many times. This is called a **loop**. For humans repetitive work is boring and they are not reliable in this. In contrast, computers are very strong and reliable in doing stupid work; they never get upset. Often it is easier to formulate a simple, easy understandable algorithm with many repetitive steps instead a complicated

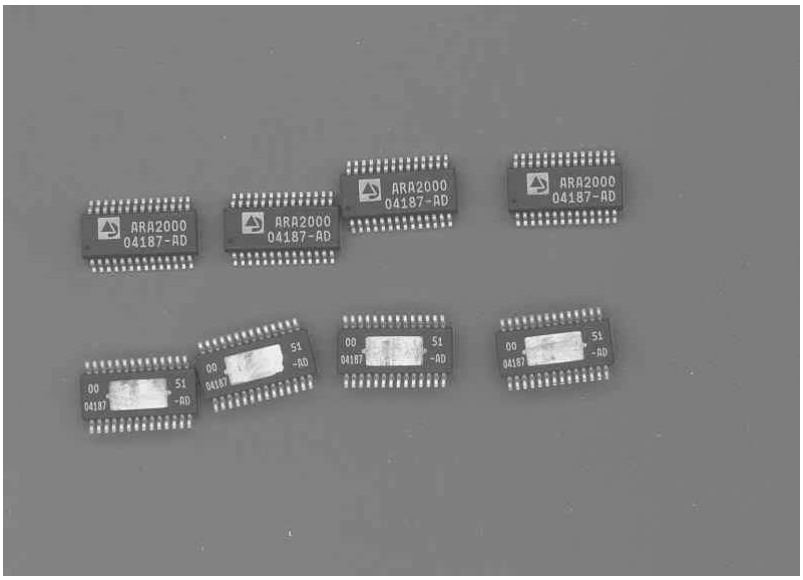

Figure 7.3 ICs as example of hardware modules

one with few steps. This choice is acceptable as long as the processing time is not disturbing.

An example for a complicated algorithm is the symbolic integration of a function. The numerical integration using the chord-trapezoid formula is reducing the problem simply to the summation of n function values. Each function value represents the area of a small rectangle that is an approximation of $1/n$-part of the function to integrate.

7.1.1 Modularization

In electronic design hardware components are used to build sophisticated devices. These components fit together because they provide a well known standardized electrical and mechanical interface. Figure 7.3 shows some integrated circuits (ICs) that have all the same mechanical and electrical interface. Therefore we can assemble these parts by plugging them into sockets to build more complex components with a richer functionality.

Software development inherently differs from mechanical construction but the idea of using "components" for constructing complex machines inspired software research.

In Computer Science an analogous concept to the ICs was developed by David Parnas who published in 1972 his famous paper about the decomposition of software systems into modules (**?**). Parnas called the software components **modules**. His idea made Software Engineering a discipline of its own.

A software *module* executes a self contained task and has a well defined **interface** to interoperate with other software components.

The **interface** of a module defines:

- the input data elements

- the output data elements

The interface does not define what happens to the input data nor how the output data is produced. In other words, a module promotes *encapsulation* – also called *information hiding* – of its internals, the data structures and the algorithms. This opens the possibility to re-implement the same functionality in a different manner without any change to the interface. It enables the programmer to replace the old module with a better one without changing or affecting the rest of the program.

We like to recall to you the subtask "read" from figure 7.2 above where this component was said to read a record in a file. Here is an example how to define its interface in Java:

```
public interface read {
  void open(String fileName) throws IOException;
  int read(char[] data) throws IOException;
  void close() throws IOException;
}
```

The interface *read* defines three methods for this "module". The *open* method takes a string that should contain the file name. It is not specified what the *open* does internally. We may speculate that the file with the given name is opened or, if this is not possible, returns with an *IOException* error. The method *read* has a character sequence parameter. The Java notation for a sequence of characters is **char[]**. Unfortunately Java does not specify if we are expected to supply some data or if the record data is returned in this buffer too. In addition, it returns an integer number. We may again speculate about the semantics of that number. The *close* does not take any parameter and returns nothing. There are better languages than Java when it comes to specifying an interface.

We note that the internal data structures, the algorithms, and implementation language are not defined for a module. This gives the programmer the freedom to implement it in any technology, with any appropriate algorithm and data structure.

7.1.2 Information Hiding

Information hiding or **encapsulation** is to conceal the design and implementation of a program or part of a program. Typical examples for encapsulation are modules and objects. We prefer the second term because the first is a misnomer: not the information is hidden, but its implementation.

Internals of a module are not exposed to the environment, and there should be no other interaction possible apart form the declared interface. This allows one to change or replace the module in case of an failure or error without affecting the rest of the software system.

In order to freely make a design decision we need to provide a stable interface which shields the usage from the details of an implementation which is subject to change. To illustrate the immunity of the interface in terms of implementation we provide the sample Java code as implementation of the *read* interface from our last example.

```
public class Reader implements read {

  private FileReader fr;
  private int pos;
  ...
  public void open(String fileName) throws IOException {
    fr = new FileReader(fileName);
    pos = 0;
  }
  public int read(char[] cb) throws IOException {
    int charsRead;
    charsRead = fr.read(cb, 0, 60);
    pos = charsRead + pos;
    return charsRead;
  }
}
```

Figure 7.4 Reader class definition

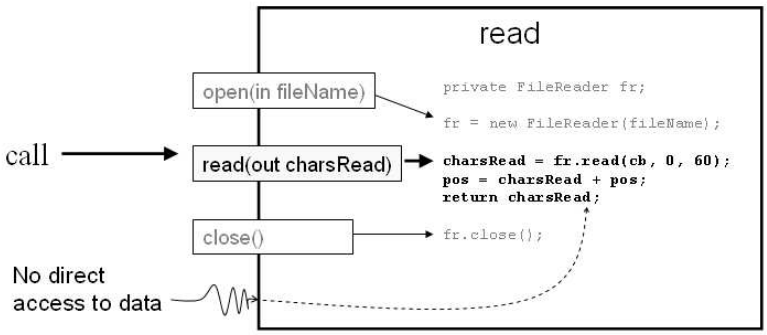

Figure 7.5 Hiding information how a file read is done in module *read*

It is not possible for the user of the class *Reader* to see that the implementation maintains *pos* and *fr* variables. The variable *pos* reflects the actual position in the file but we have no chance to access this value because the variable is hidden by the name *private*. In fact this information is nowhere returned to the user of that class. The *fr* variable was assigned to identify the file in method *open*. It is reused in the method *read*. But again, this information is hidden from the user of the module. The situation is illustrated in figure 7.5. In the context of object oriented programming the concept of information hiding is called *encapsulation*.

Assume we find a better way to implement the *read* method. For example as the value of *pos* is nowhere used, we could reimplement the method *read* in a more compact way:

```
public int read(char[] cb) throws IOException {
  return = fr.read(cb, 0, 60);
}
```

This will not affect the formal use of our module as long as the interface remains unchanged. Information hiding reduces the risk of misusing the functionality of a module and its implementation dependencies.

However, the interface did not tell us that we can read at most 60 characters in one call. The documentation of the methods should tell us this important and limiting fact. If we decide to change this limit the result is a different semantic of the *read* method which can entail changes in programs using this module. We see from this example that it is not easy to maintain Java modules without any side effects.

Another problem with our *Reader* example is the dependency of the method *read* on the method *open* which has to be called before *read* can be used.

A much more powerful specification language than the Java interface definition will be needed to specify the semantics of an interface.

7.1.3 Object Orientation

With the deeper understanding of information hiding and modularization the object-oriented paradigm arose in the 70^{th} that boosted software development methodologies.

Following the *object-oriented paradigm* we perceive the world and their programmed representation as interacting objects. An **object** is a self-contained entity like a module that consists of both data (called **state**) and operations (called **methods**) to manipulate the data. The data inside these objects is not visible (*information hiding*) and cannot be manipulated directly from outside because it is *encapsulated*.

Objects interact with each other by sending *messages*. Sending a message to an object causes the execution of a method at the receiving object. A *method* is similar to a function in procedural languages. Nevertheless, the conceptual difference between both is that the caller of a function decides which function is activated but in object-oriented environments the receiver of a message decides which activity (method) is accomplished. Thus objects can react differently to a message depending on the object's type or state (see Subsection 8.2.4 in the next chapter).

Objects with the same behavior are said to be of the same **type**. A **class** is the implementation of a type, i.e. a class defines the internal structure of a type and implements its behavior in form of methods. In addition, the class defines constructors how to create new objects. Objects are therefore called *instances* of its class.

As example let us take again the administration of students. We consider the students as objects. Each student object contains data like the name, address, courses taken and examination results (see Figure 7.6). The common structure and functionality for all student objects is defined in a class named "Student". An new student record corresponds to the creation of a new object. A student's record is read or updated by sending the corresponding message to the object.

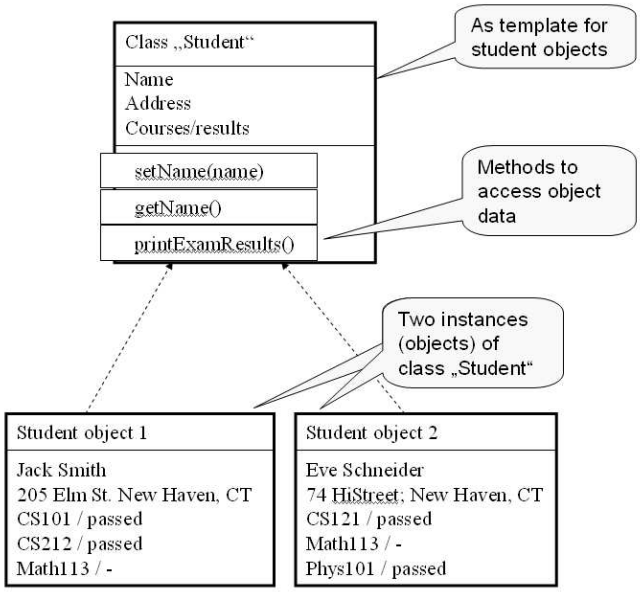

Figure 7.6 Student objects

To produce a list of a student's examination results just ask the student's object to do so by sending a *printExamResults* message. Provided the corresponding method exists, it will produce the list and send it to the printer object who will print the information on paper.

We will learn more about object oriented features in conjunction with object oriented programming languages in the next chapter.

7.2 Module Coupling

Coupling – in the sense of software modules – deals with the dependency between software units. The coupling is tighter if more dependencies for a module exit. A "loose" coupling is desirable because changes of a module are less likely to affect other modules, i.e. the modules are more independent of each other.

7.2.1 Adhesion

In software development **adhesion** is a measure of how tightly a software module is depending on other modules. The more dependencies exist the stronger the adhesion to other modules. The more (complex) parameters the modules need to work the more tightly is that module "glued" to its caller. A measure for the explicit coupling of a module to its caller are the number and the complexity of its parameter. Even worse are implicit dependencies by global variables or other environmental context.

Take a module that stores properties like the name, date of birth, or phone number of a person. If we do not identify the person's object, the module cannot know who's name or phone number to store.

If the module has to take the person's data from a global variable, this is a dependency that inhibits the reuse of the module in a different context where such a variable does not exist. It is much better to pass the person's object as a parameter to the module or incorporate the person's data in the module itself (see Figure 7.7).

It is desirable that a module has the least possible adhesion. This means that the module should not depend on other software, i.e it should be self sufficient. That means that a module should be context free. Modules with the least possible adhesion are objects that do not depend on objects outside the module.

In modern web service applications this is a key aspect. Web services usually require a context free task with simple structured parameters.

7.2.2 Cohesion

In computer science the word **Cohesion** describes how well a module serves a task. It describes the extend of internal cooperation within a module. Cohesion is an ordinal measure that is expressed as a number or qualified as "high" or "low". In a module with high cohesion all program lines work together to perform the task. Low cohesion signifies that only part of the module is needed.

A module should serve only one logical task. If it serves more tasks, only part of the program code is needed for performing a particular task.

A module with high cohesion is software that represents exactly one entity and serves only one task. Modules with high cohesion tend to be reusable, robust, and easy to understand.

Take a module that stores the name, date of birth, and the phone number of a person. If a program wants to use this module it has to provide values for all three parameters. But you cannot use the module if you only want to update the phone number unless you supply the name and date of birth as well.

This problem arises from the fact that the module tries to combine three functionalities in one module. It would be better to have three modules, one for each function.

Cohesion is often correlated with adhesion. Modules with a high cohesion tend to have less adhesion and therefore less coupling dependencies.

In order to develop versatile and reusable modules try to get a high cohesion and low adhesion as in Figure 7.7. The complexity introduced by the adhesion of modules in a system tends to increase with the number of modules. At the same time the cohesion reduces the complexity if the system is more modular. The goal is to find the optimal number of modules for a system with minimal complexity. This is shown in Figure 7.8.

7.3 Repetition

One of the most widely used programming habits is the repetitive use of a program code sequence. This is a tempting concept because it is easy to program and the computer is fast and very reliable in performing repetitive tasks.

Computing exactly the same instructions again would produce the same result. This is worthless because it produces no new result. Therefore only if

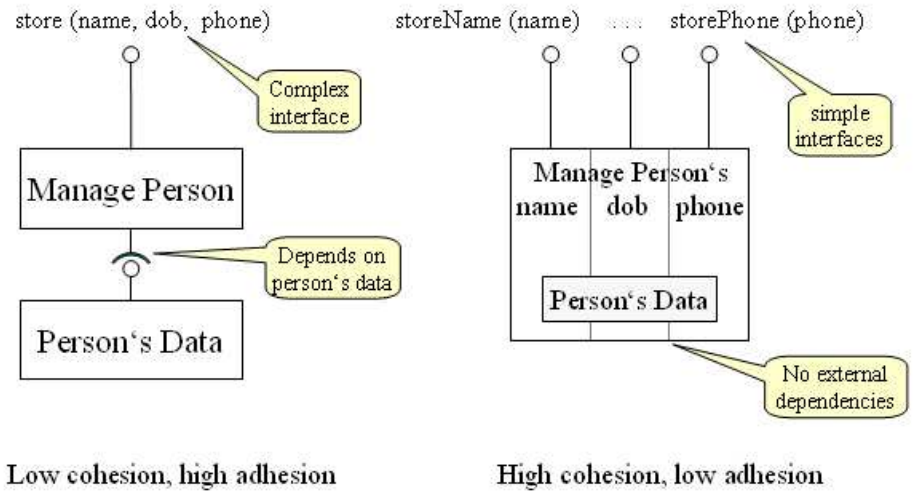

Figure 7.7 Example for high and low cohesion and adhesion (coupling)

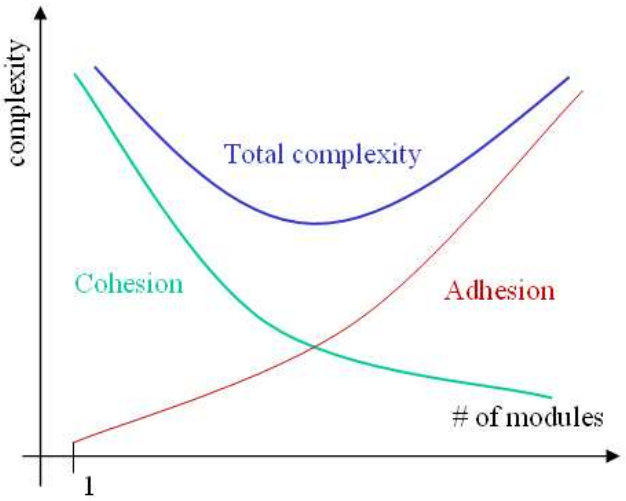

Figure 7.8 Complexity of a system as function of the number of modules

there is a (little) change to the data each time the execution makes sense. There are *two* techniques to change the data when the same program instructions are repeated. The *recursion* technique calls the same algorithm again within itself using a different parameter value.

The *iteration* (also called "looping") re-executes a sequence of instructions with different data for a number of times.

In many cases iteration can be reformulated as recursion which looks more compact and elegant. But, for programming the iteration is more efficient.

The difference between both repetition techniques is that *iteration* always executes the complete instruction sequence before re-execution and *recursion*

executes only one part then calls itself again. The rest of the instructions are executed when the nested call returns.

7.3.1 Recursion

Some types of algorithm refer to itself, i. e. build up a **recursion** stack. The classical example is the factorial function where $fac(n)$ is reduced to $n \cdot fac(n\text{-}1)$:

$$\begin{aligned} fac : \mathbf{N_0} &\rightarrow \mathbf{N} \\ n &\mapsto fac(n) \end{aligned}$$

where the function fac is recursively defined as:

$$\begin{aligned} fac(0) &:= 1 \\ fac(n) &:= n \cdot fac(n-1) \text{ for } n \geq 1 \end{aligned}$$

Recursive definitions need a termination point. In our example, this is given by $fac(0) := 1$.

When a factorial, say $fac(3)$, is computed the following steps are executed:

$$\begin{aligned} fac(3) &= 3 \cdot fac(2) \\ fac(2) &= 2 \cdot fac(1) \\ fac(1) &= 1 \cdot fac(0) \\ fac(0) &= 1 \end{aligned}$$

resulting in $fac(3) = 3 \cdot 2 \cdot 1 \cdot 1 = 6$. Clearly, in this example the termination point is $fac(0)$.

As each step computes a factorial, we can shorten the algorithm to:

$$fac(3) := fac(fac(fac(fac(0))))$$

This makes clear that recursion is repeating a *similar* task *within* a task. This is like a Russian Doll where each doll contains a smaller doll of the same kind.

To show how a recursion is programmed in a procedural style in Java we list the method *factorialRecursive()*:

```
long factorialRecursive(long n) {
    if (n > 1)
        return n· factorialRecursive(n − 1);
    else
        return 1;
}
```

When the method $fac(n)$ is executed the method is n-times activated before the first result value (1) is available. The computer has to hold the environment

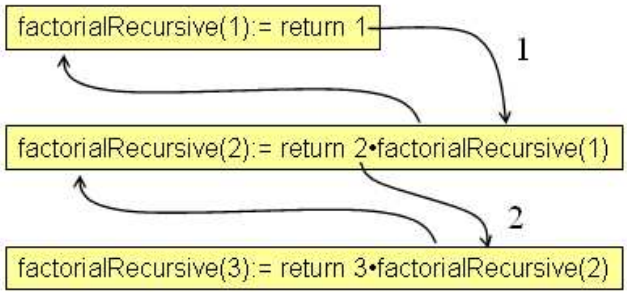

Figure 7.9 Visualization of the factorialRecursive(3) recursion stack

of n active fac-methods and then give the result successively back to its calling method and multiplying it there with the next number until all methods have finished.

If we call $factorialRecursive(3)$ we get the execution stack visualized in Figure 7.9

The memory used and organizational overhead in our *factorialRecursive* example is much more than doing a multiplication n times. The environment of each function call needs to be kept until all following calls have returned their results. From the efficiency point of view a "looping" algorithm for the factorial is much better the the recursive.

In general, recursive algorithms are more compact and elegant than other solutions. With certain dynamic data structures (e.g. trees) only the recursive solution is adequate.

7.3.2 Iteration

If the same operations are needed repeatedly to approach an solution this is called an iterative solution. The repetition is technically obtained by "looping" over the operations.

The factorial example is easily formulated with an iterative algorithm:

$fac := 1$
for $i := 1$ to n step 1 do
  $fac := fac \cdot i$
end for loop
return fac

The "for-loop" is not executed if the upper limit n is less than the start value (1 in our example). In this case fac returns the initial value 1.

Iteration is often more efficient than the corresponding recursive algorithm. But recursion is the more elegant and more descriptive way of formulating an algorithm.

Again we show the algorithm for *factorialIterative()* in Java source code:

```
long factorialIterative(long n) {
    long fac = 1;
    for (long i = 1; i <= n; i++) {
        fac = fac · i;
    }
    return fac;
}
```

Here the variable *fac* is used to hold the intermediate results for the factorial. Please note the difference to the recursive approach where the factorial function itself is used to compute partial results.

In a stricter sense *iterative algorithm* means that the solution is approached step-by-step. The solution starts with a rough guess or an arbitrary value. Each iteration step attempts to find a better approximation for the solution. The process is stopped when two consecutive approximations differ less then a predefined value ϵ.

The $\cos(x)$ function is used to illustrate the use of ϵ. The following series expansion for $\cos(x)$ is well known in Mathematics:

$$\cos(x) = \sum_{n=0}^{+\infty} \frac{(-1)^n x^{2n}}{(2n)!}$$

If the cosine is computed this way, the algorithm will never terminate. The sum is approaching its limit slowly as n grows. The exact value for $cos(x)$ lies between two consecutive approximations. Stopping the calculation yields an error less than the difference of two consecutive approximations. Given an $\epsilon > 0$ we can stop adding terms when

$$\frac{x^{2n}}{(2n)!} + \frac{x^{2(n+1)}}{(2(n+1))!} < \frac{2x^{2n}}{(2n)!} < \epsilon.$$

Take as example $x = 1$ and $\epsilon = 10^{-6}$, then only $n = 5$ terms are needed to sum up to get the result for $cos(1)$ with an accuracy of 10^{-6}.

This example serves as accuracy tradeoff for a predefined accuracy ϵ. For a fixed number n this is also an example for a time tradeoff with predefined computation time.

7.4 Design Patterns

Design patterns are not an invention of computer science but have been known since long in construction and craftsmanship. Well known re-applicable solutions to certain types of construction problems have been taught and used by generations of engineers. Architects used arches to make sure that a bridge could carry heavy load and a ceiling is withstanding its payload.

Engineers used toothed wheels for slippage-free transmission of rotation and gears to adjust rotation speed.

Computer scientists transferred these technical cognitions to the design of software. In the year 1995 Erich Gamma and others published a book with about

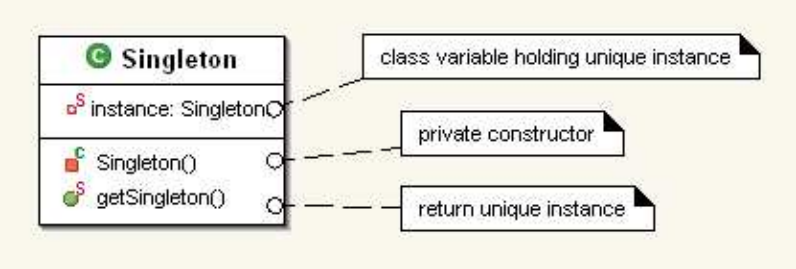

Figure 7.10 UML class diagram for the singleton pattern

two dozen of *software design patterns* which made reusable design solutions popular in software engineering.

The design patterns are divided into three categories:

- creation patterns
- structural patterns
- behavioral patterns

In the book of Gamma et al (**?**) each pattern is presented in a strongly structured way, explaining the *intend, motivation, application, structure* with classes and interactions. The description lists *consequences* and gives *implementation* examples. To give an impression of this approach we will present some example patterns for each category in the same manner.

7.4.1 Singleton

This pattern belongs to the creation category. Its *intend* is to make sure that a class has exactly one instance.

The *motivation* for this is that for certain classes it is important to have only one instance to make sure that access to a resource is done via one object only. How is it possible to ensure that only one instance is available? A global variable may provide an access point, but cannot guaranty that only one instance will be created. It is therefore necessary to protect the constructor and that its use is limited to only one instance creation.

The UML diagram (Figure 7.10) shows the *structure* and the methods for the singleton class.

The singleton pattern is used if exactly one instance of a class must exist with a well defined access point. Another *application* for a singleton is to extend the single instance by subclassing without changing the interface.

The *consequences* of a singleton pattern are:

- full access control to the singleton instance
- avoids the excessive use of global variables
- a singleton may be subclassed with refined operations
- extension to a controlled number of instances is possible

Let's now discuss some important *implementation* aspects. The singleton pattern must make sure that only one instance can be created and that this instance is a normal instance of its class. Usually the instance is created by a class method. That method must have access to a class variable that holds the only instance of the singleton class. The implementation in Figure 7.11 uses lazy initialization of this class variable. This technique simply checks if the variable is initialized before it retuns its value. If the variable is "null", the instance is created and then returned to the caller.

```
public class Singleton {
  private static Singleton instance; // only internal access to class variable
  private Singleton() { // private constructor
  }
  public static Singleton getSingleton() {
    if (instance == null) { // lazy initialization
      instance = new Singleton();
    }
    return instance;
  }

}
```

Figure 7.11 Coding example of a singleton class

7.4.2 Factory method

An other creational pattern is the factory method. Its *intend* is to create objects, but delegate the creation process to subclasses. Simply speaking, the pattern defines an interface for subclasses that decide which objects to create.

The *motivation* for this pattern is that applications need to create objects at runtime that are not known at programming time. The application only knows the abstract superclass where no instances can be created. The factory pattern solves the problem by encapsulation of the creation process into subclasses of the abstract superclass.

There are many *applications* for this pattern. Nearly every framework has to create and manage objects that cannot be foreseen by the framework developer. It is used when a class

- does not know the object to create in advance
- wants to delegate the creation to subclasses
- wants to localize the knowledge about the creation process.

The *structure* consists of two class hierarchies, the product classes and the factory class hierarchy that will create the product instances. Usually the product instances are components of the application that creates the instances. Each concrete factory class depends on the concrete product classes it creates.

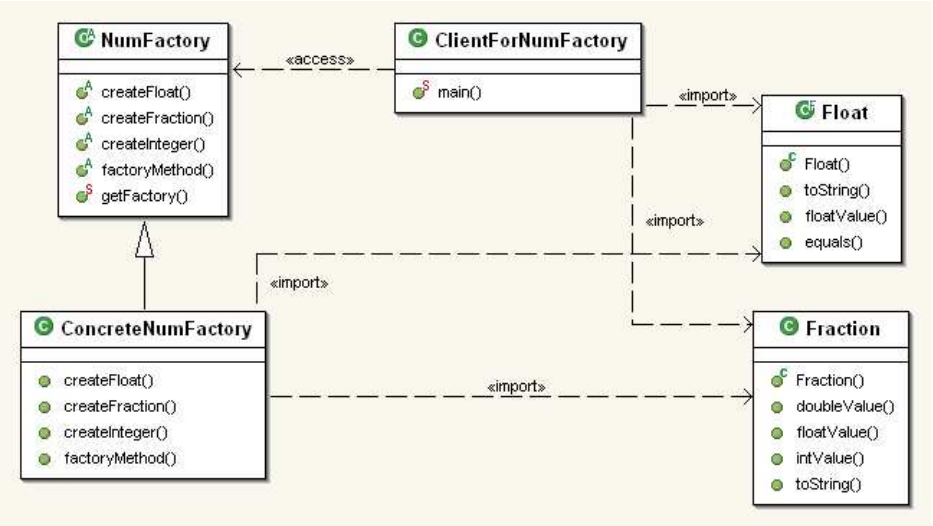

Figure 7.12 Class diagram of a Number factory method pattern

An example of the factory pattern is shown as UML class diagram in Figure 7.12. It demonstrates how to create numbers in a uniform way without knowing the exact instantiation mechanism. Only the factory method must be known and the concrete instantiation is forwarded to the actual number class.

An *implementation* of this example is presented in Figure 7.13. The *ConcreteNumFactory* class defines factory methods for three types of Number: Integer, Float, and Fraction. The actual instance creation is implemented in the number subclasses. The source code for *Fraction* class is left as an exercise for the reader. The other classes, Integer and Float are defined in the Java class library, called Java Development Kit (JDK).

The application *ClientForNumFactory* in Figure 7.14 uses the factory pattern to create concrete numbers of type Integer, Float, and Fraction. Because the creation process is different for different classes, the application cannot know the creation process in advance. The application only instantiates a *NumFactory* and leaves the creation to the subclass *ConcreteNumFactory*. In our application example *ClientForNumFactory* we have three factory methods, one for each Number subclass. With this technique the application does not need to know how to create a specific Number (say Complex number) that might be defined later. This special knowledge could be implemented in a new subclass of *NumFactory*.

In the next subsection we present three important structural patterns.

7.4.3 Composite

The *intend* of the composite pattern is to provide a structure to represent aggregate-part hierarchies where parts or components can be handled in a uniform way.

Configuration of a complex product like a PC in an order entry system needs to compose the product with its parts and components. The order system does not know the actual composition in advance. Therefore it is not possible

```
public class ConcreteNumFactory
extends NumFactory {
  public Integer factoryMethod(int i) {
    return new Integer(i);
  }
  public Float factoryMethod(float f) {
    return new Float(f);
  }
  public Fraction factoryMethod(int z, int n) {
    return new Fraction(z, n);
  }
}
```

Figure 7.13 Coding example fragment using the factory pattern

```
// how to use the factory
public class ClientForNumFactory {
  public static void main(String[] args) {
    NumFactory fact = NumFactory.getFactory();

    Integer i = fact.factoryMethod(5);
    Float f = fact.factoryMethod(5.9f);
    Fraction q = fact.factoryMethod(1, 3);

    System.out.println("Create Fraction(1/3) . . . = "+q);
  }
}
```

Figure 7.14 Coding example fragment using the factory pattern

to hard-code all possible configurations. Here comes the composite pattern into play because it solves this design problem by delegation. The *motivation* for this pattern is apparently to handle each part in a specific way even if the user does not distinguish. For instance different parts of the PC are plugged into the case and cabled in an interface specific way, but the user only needs to drag-and-drop the part even if different assembly methods apply to each part.

Applications for the composite pattern are:

- whole-part hierarchies
- ignore differences in parts of an aggregate
- uniform access to different parts

The aggregate-part hierarchy is represented by the composite pattern as object containment hierarchy. The general *structure* for a composite is shown in Figure 7.15. The pattern uses an inheritance hierarchy to ensure the same operations for all kind of components. An object may represent a single part (called leaf)

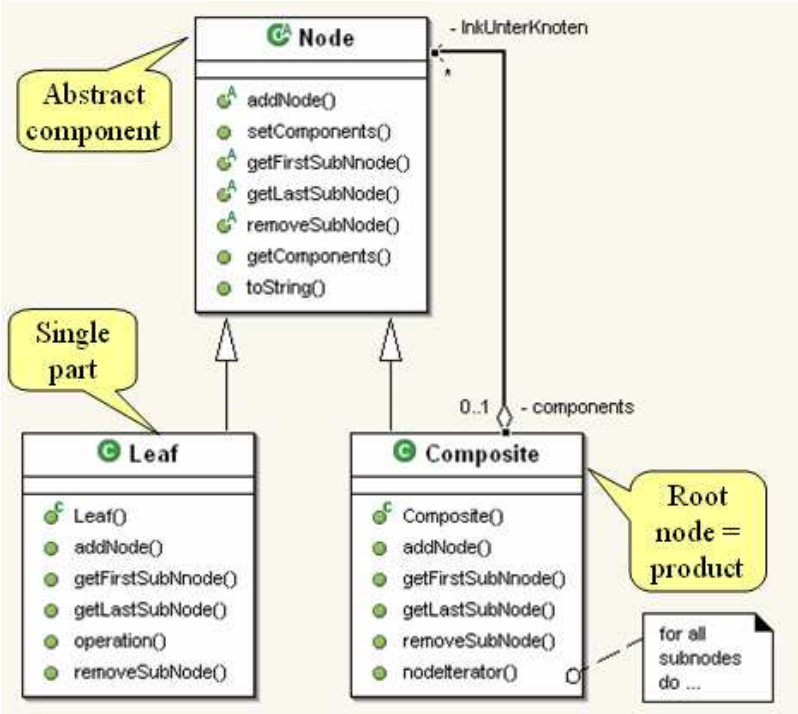

Figure 7.15 Class diagram for the composite pattern

or a composite part which contains other components. This defines a recursive structure forming a tree (see Section 4.4) where the root object is called the "whole" or the product and the objects at the "bottom" – the leaf nodes – are the single parts. All nodes in between are composite nodes that belong to exactly one parent object (component) and consist of one or many parts or components (the "children"). The parts contained in an aggregate are ordered in a specific way to build it up. This allows to navigate the tree structure with methods like *getParent()*, *getFirstChild()*, *getNextChild*, *getPreviousChild()*, etc.

For an *implementation* we refer to the navigation and searching examples of tree structures in Chapter 5 that have the same structure and operations.

7.4.4 Decorator

The decorator's *intend* is to dynamically extend the functionality of classes. It is also known as **wrapper** pattern.

There are situations where we want to add functionality to a class without subclassing. Subclassing is not flexible enough to change dynamically the functionality of a class. The *motivation* is to enclose (wrap) an object by an other object providing the same interface as the enclosed object plus some more operations. Wrappers may be stacked like Russian dolls so that the functionality can be extended gradually and multiple times.

The pattern is used for *applications* that:

- dynamically add and remove functionality of an object

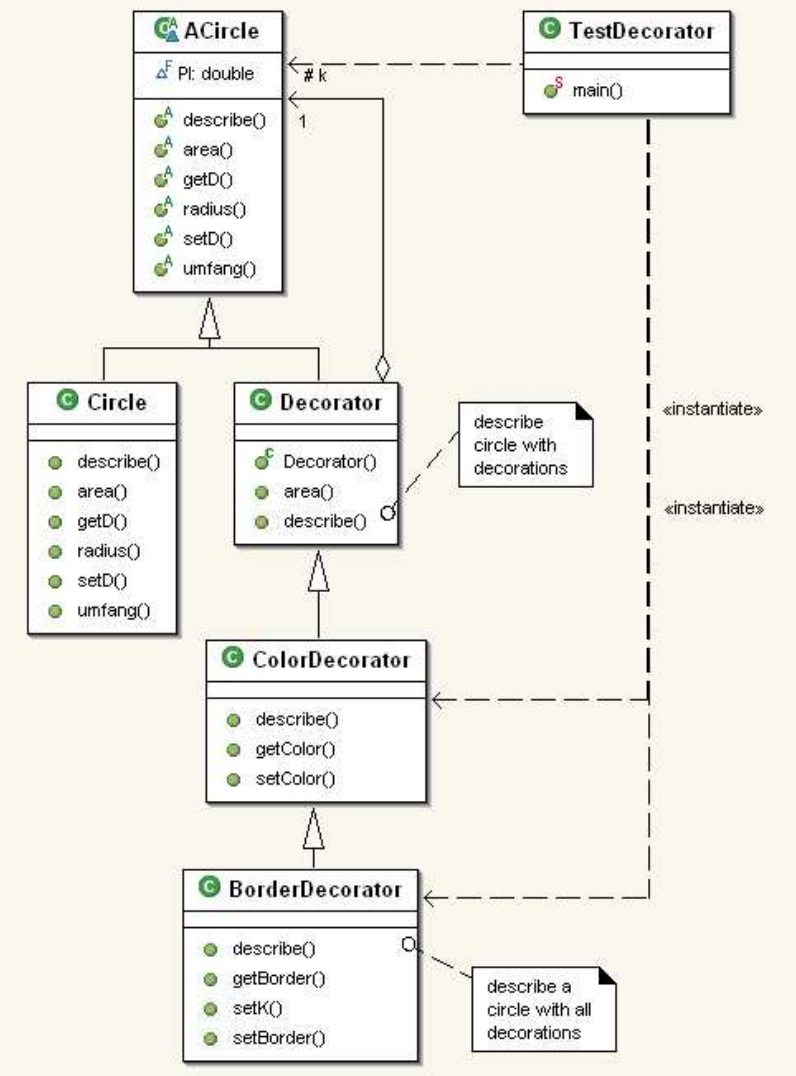

Figure 7.16 Decorator class diagram

- cannot use subclassing because it would need too many subclasses

The decorator and the composite pattern both use delegation to either extend or use specific functionality. The *structure* of the decorator (see Figure 7.16) resembles the composite pattern but the multiplicity of the components is always one. This degenerate tree is in fact a stack of decorators with the last decorator enclosing the object. An instance of a decorator stack is shown in Figure 7.17. Please note, that the border decorator class *BorderDecorator* uses the color circle decorator class *ColorDecorator* resulting in a circle with color and a border. Alternatively the *BorderDecorator* class could inherit directly from *ACircle*. In this case the *ColorDecorator* does not need to exist when the *BorderDecorator* is defined.

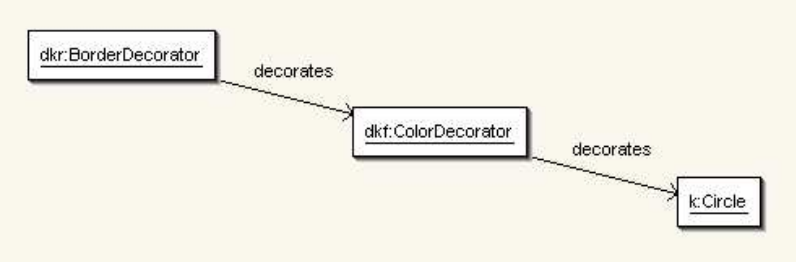

Figure 7.17 Example instance of a decorator pattern

As *implementation* example let's take the *Circle* class and "decorate" it with a color and a border line. The decorator pattern dynamically adds specific operations like setting a color or defining a border line width, etc. for the "decorated" circle.

The example includes an abstract *ACircle* class (see Figure 7.18) that is the common superclass for the concrete circle and the decorator that augments the circle with color and a border. Figure 7.19 shows how the *ColorDecorator* defines getter and setter methods for the color. The method that describes the colored circle is realized by overriding the inherited method.

The *BorderDecorator* is not listed because it has a similar code structure but decorates the ColorDecorator instead of the circle.

7.4.5 Adapter

An adapter is a decorator that changes the interface in order to "adapt" it to its user. The *intend* is evident, the adapter adjusts an class interface to the interface that is expected by its user. It makes two classes compatible to communicate with each other.

The *motivation* to use an adapter results from the following situation. An application should be extended and the required functionality is already provided by an existing class, but its interface is incompatible with that application. The adapter makes the class compatible with the application without changing neither the class nor the application.

Use the adapter pattern

- when an existing class is used that does not have the required interface
- to develop a reusable class that should work with classes that do not have compatible interfaces
- for existing subclasses if you do not want to adapt the interfaces of each subclass

The *structure* distinguishes two adapter types (see Figure 7.21)

- adapter uses multiple inheritance (class adapter)
- adapter uses object composition (object adapter)

```
abstract class ACircle {

  final double PI = 3.14159;
  protected double d;

  abstract public double radius();
  abstract public double umfang();
  abstract public double area();
  abstract public double getD();
  abstract public void setD(double dd);
  abstract public void describe();
}
public class Decorator extends ACircle {
  protected ACircle k;
  public Decorator() {
    k = new Circle();
  }
  public double radius() {
    return k.radius();
  }
  public double area() {
    return k.area();
  }
  // more methods
  public void describe() {
    k.describe();
  }
}
```

Figure 7.18 Coding example for a decorator pattern providing color for a circle (part 1)

From the user's (called client) point of view there is no difference. The difference lies in the *implementation* of the adapters only. An implementation example for an object adapter is given in Figure 7.23. The code shows how to adapt an integer object to work with floating point objects. For simplicity only the floor() method is adapted. The reason for using the "home made" float class *MyFloat* is that Java defines the *Float* class as "final". This property forbids further subclassing. The corresponding class diagram is presented in Figure 7.22. The diagram shows the typical object adapter structure where class *FloatAdapter* inherits from *MyFloat* to get its interface and uses (called <<import>> in the UML diagram) an instance of *Integer* to adapt the integer object.

Figure 7.24 gives an example how to use the *FloatAdapter* adapter pattern. It adapts the *foor()* method that returns the largest integer less or equal the number value. This function is well known for Float numbers and is trivial for Integer numbers. With this adapted method it is possible to treat Float and Integer numbers uniformly with respect to the floor operation.

```
public class ColorDecorator extends Decorator {

  private String farbe;

  public String getColor() {
    return farbe;
  }
  public void setColor(String f) {
    farbe = f;
  }
  public void describe() {
    super.describe(); // describe circle
    System.out.println("Color = "+this.getColor());
  }
}
```

Figure 7.19 Coding example for a decorator pattern providing color for a circle (part 2)

Object adapter make it easy to work with more than one class. In fact if a new subclass is added, the adapter works automatically with this class. An other *consequence* is that object adapter makes it difficult to overwrite the behavior of an adapted class. The class adapter has exactly the opposite advantages and disadvantages. It adapts exactly one class only. Therefore it is easy to overwrite the behavior of an adapted class.

The next patterns are part of the original eleven behavioral patterns of the classical *Design Pattern* book **?**.

7.4.6 Observer

The *intend* of this object based pattern is to define an one-to-many dependency between objects. All dependent objects are notified and updated when the independent object changes its state.

The *motivation* of this patter arises from the need that all representations and visualizations of an object's data should be consistent. If a change happens, the new situation should be propagated to all objects using or depending on that data. An *application* for the observer pattern is the **model-view-controller** paradigm that *uses* the pattern to synchronize all views with the model.

The model and view classes are called "loosely coupled" because each one can be used without the other. In particular two views do not know from each other even if they look like working together. The synchronized behavior is realized via the model object as shown in Figure 7.25. Any change in the data of the model object is signalled to all observers. They in turn must then read the new values and update their visualizations.

Our example *implementation* uses the notification and thread mechanism of Java. The structure of the application is given in Figure 7.26. The two observers *DigitalDisplayN* and *AnalogDisplayN* implement the Observer interface *update()* to read the time value. They are subclasses of *Thread* a library class from the Java Development Kit (JDK) that provides the functionality to run each class

```
public class TestDecorator {

  public static void main(String[] args) {

    Circle k = new Circle();
    k.setD(10);

    ColorDecorator dkf = new ColorDecorator();
    dkf.setColor("blau");
    dkf.setK(k);

    BorderDecorator dkr = new BorderDecorator();
    dkr.setBorder(2);
    dkr.setColor("rot");
    dkr.setK(k);

    System.out.println("Circle: " );
    k.describe();
    System.out.println("Color-Decorator: " );
    dkf.describe();
    System.out.println("Border-Decorator: " );
    dkr.describe();
  }
}
```

Figure 7.20 Coding example fragment for a decoration pattern providing three stream types

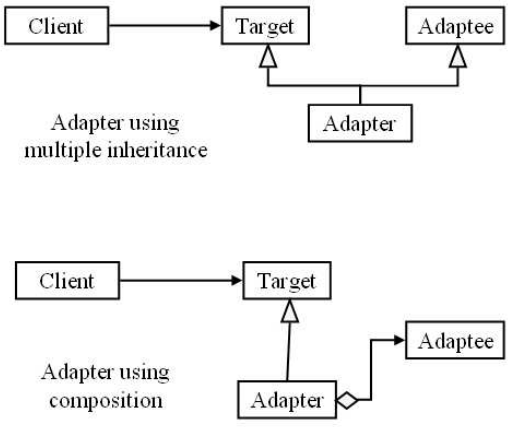

Figure 7.21 UML class diagrams for adapter pattern

as a separate thread of execution. This ensures that the views run independent of each other. The model is played by the class *Movement* which represents the

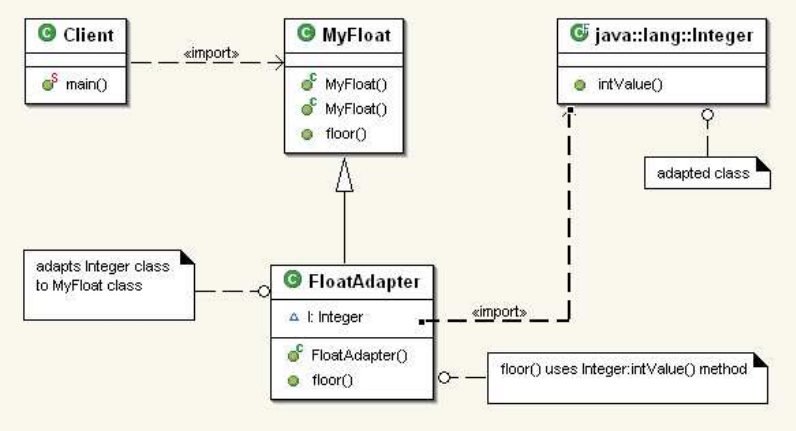

Figure 7.22 UML class diagram for FloatAdapter

```
public class MyFloat {
  protected Float F;
  public MyFloat(float value) {
    F = new Float(value);
  }
  public double floor() {
    return Math.floor(F.floatValue());
  }
}

public class FloatAdapter extends MyFloat {
  Integer I;
  public FloatAdapter(int value) {
    I = new Integer(value);
  }
  public double floor() {
    return Math.floor(I.intValue());
  }
}
```

Figure 7.23 Coding example for adapting the Integer to Float class protocol

clockwork. The notification is done by the *notifyAll* method that notifies the observers. Internally the notification mechanism maintains a list of all observers (displays). This enables the model to signal all dependent displays the time change. No matter if the view is an analog or digital display, the time shown should be in synchronization with the clockwork. After the notification it is the responsibility of the observers to read the change and update the displays.

The source code for the clockwork model class *Movement* is listed in Figure 7.27 and one concrete observer, the *DigitalDisplay* class, is listed in Figure

```
public class Client {
  public static void main(String[] args) {
    System.out.println("Adapter Test . . . ");
    MyFloat f1 = new MyFloat(5.9f);
    System.out.print("MyFloat(5.9f) understands floor() = ");
    System.out.println(f1.floor());

    System.out.print("floor() is undefined for Integer(5)");
    System.out.println(" but");

    FloatAdapter fa = new FloatAdapter(5);
    System.out.print("FloatAdapter(5) understands floor() = ");
    System.out.println(Math.floor(fa.floor()));
  }
}
```

Figure 7.24 Coding example for adapting the Integer to Float class protocol

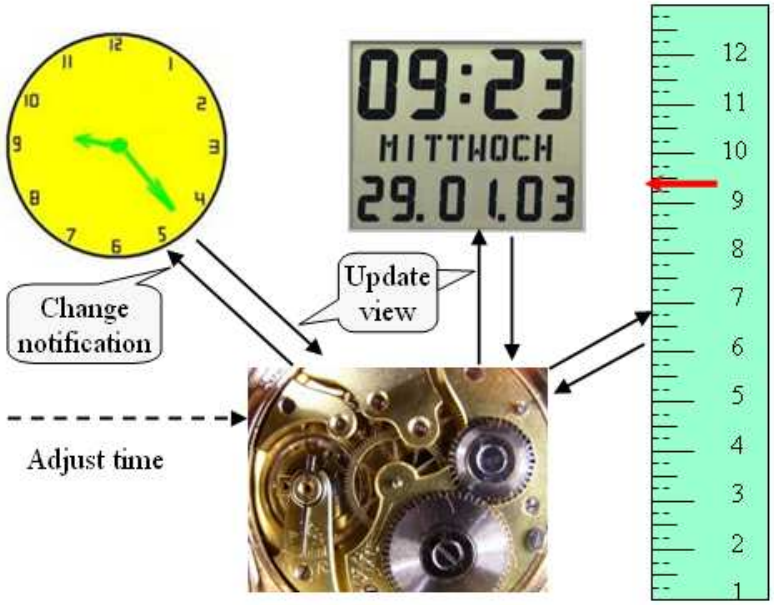

Figure 7.25 Clockwork with different displays as example of the observer pattern

7.28. The implementation of the *AnalogDisplay* class is left to the reader. The movement reads every second the system time and notifies the observers to update their views. The observers use the *update()* method to read the actual time from the variable *ticktack* by sending the message *getTime()* to the movement object (*clock*). Calling system functions like waiting for some seconds or waiting until notified may result in an exception that has the programmer to be aware of. In the demo program the exception is ignored resulting in a missed

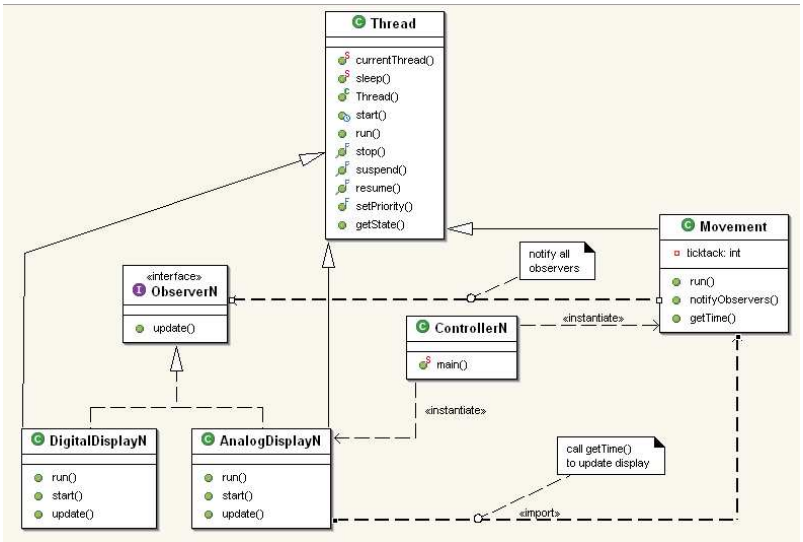

Figure 7.26 UML class diagram for an example observer pattern (clockwork)

update. The display returns to sync when the next notification (after another second) occurs. Notification and updating the time is done in a *synchronized* block to make sure that the access is mutual exclusive.

The class *ControllerN* gives an example how to use and work with the observer pattern. The source code shown in Figure 7.29 is extremely simple: it is sufficient to create the model (movement) and the observers (analog and digital display), then start each as a separate thread. Most of the work is done in the *run()* instance methods of the model and observer classes (see Figures 7.27 and 7.28). The *start()* method is inherited from the superclass Thread and *run()* has to be implemented individually to reflect the specific functionality.

Consequences of this implementation are that the model has a list of all objects that provide the observer interface. But the model does not know in which sequence and how the observers are informed. This is the task of our last behavior design pattern that we present next.

7.4.7 Iterator

The *intend* of an iterator or **cursor** pattern is to access the elements of a list, tree, set, or composite. All these collection types should provide a mean to access all elements without exposing its internal structure.

This *motivates* the iterator pattern that let's you traverse the elements in a uniform way by separating the access operations from the collection or composite structure. We call such an iterator concept polymorphic if it provides the same access-interface for different object types.

The pattern is *applied* to

- access the elements of a composite object
- travers a collection multiple times
- provide a polymorphic interface for different collection structures

```
public class Movement extends Thread {
  private int ticktack = 0;
  public void run() {
    while (true) {
      ticktack = (int)(System.currentTimeMillis()/1000);
      synchronized(this) {
      this.notifyObservers();
    }
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e){ };
    }
  }
  public void notifyObservers() {
    this.notifyAll();
  }
  public int getTime() {
    return ticktack;
  }
}
```

Figure 7.27 Implementation example of observer pattern (model part)

The iterator is built as a *structure* of two class hierarchies. One is the aggregate or collection hierarchy that creates the iterator hierarchy. The example structure in Figure 7.30 shows *AbstractList* and *Vector* as the collection hierarchy. The iterator hierarchy is composed of the interface *Iterator* and its implementation *Itr* which is an embedded class.

This structure has three *consequences*:

- variants for the traversal are possible
- the iterator simplifies the aggregate interface
- the aggregate may be traversed simultaneously by two or more iterators.

The *application* in Figure 7.31 shows how to use the iterator pattern provided by the JDK. The Vector class supports an instance method *iterator()* that returns an iterator for the vector object *v*. With this iterator it is possible to access each element in sequence. The method next() returns the next element. What is "next" defines the iterator. The internal structure of the vector object is not exhibited. An other useful method is the *hasNext()* method that returns true if there are more elements to access.

The *select(opCode, value)* method for the Vector *v* selects different types of numbers satisfying a comparison expression. The method handles all kind of numbers in a polymorphic way. The same iterator pattern would work if we replace the Vector by any other collection class e.g. say a HashMap.

```
public interface ObserverN {
  abstract void update(Movement clock);
}

public class DigitalDisplayN extends Thread implements ObserverN {
  private Movement clock;
  public void update(Movement clock) {
    System.out.println("Timestamp = "+clock.getTime());
    try {
      clock.wait();
    } catch (InterruptedException e) { }
  }
  public void run() {
    System.out.println("run() until ctrl-brk");
    while(true){
        synchronized(clock) {
        this.update(clock);
      }
    }
  }
  public void start(Movement c) {
    start();
    clock = c;
    System.out.println("DDisplay started with clock = "+clock);
  }
}
```

Figure 7.28 Implementation example of observer pattern (observer part)

```
public class ControllerN {
  public static void main(String[] args) {
    Movement c = new Movement();
    AnalogDisplayN a = new AnalogDisplayN();
    DigitalDisplayN d = new DigitalDisplayN();
    c.start();
    a.start(c);
    d.start(c);
  }
}
```

Figure 7.29 How to use the observer pattern

But, you may have noted, that we do not use the iterator for the *Number[]* array. Instead we use a “for-loop” construct. The reason for this is a flaw in the implementation of Java. Apparently the array is not a subclass of the AbstractCollection that provides the iterator pattern.

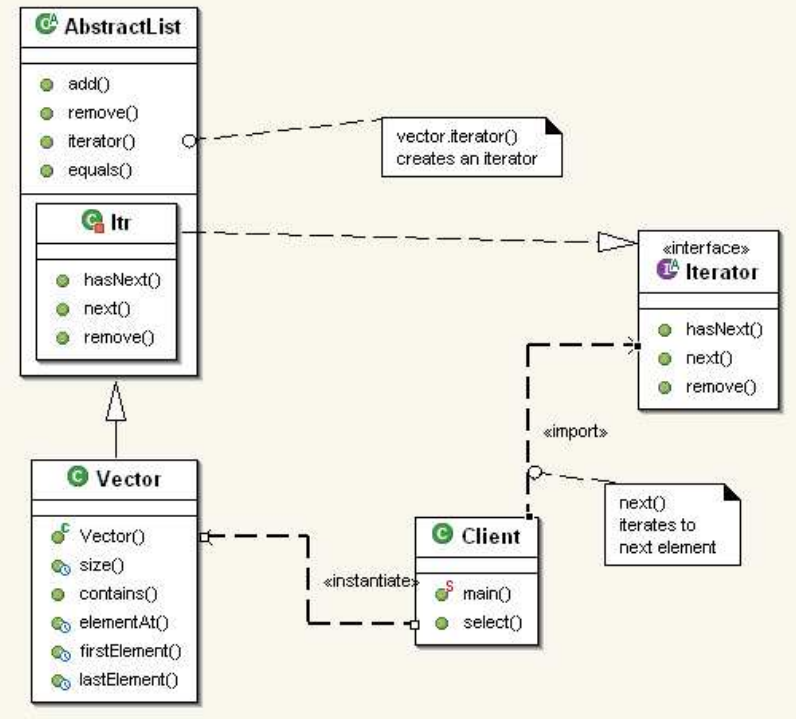

Figure 7.30 UML class diagram of the iterator pattern

7.5 Summary

Exercises

7.1 How would you refine the task *delete*? Could the parts *read*, *display*, *edit*, *write* be of any use?

7.2 What happens when the argument n of the function *factorial* is zero or negative? Is this in accordance with the definition above? Change the method to return an error when the argument is negative.

7.3 Point out the differences of hardware compared with software.

7.4 How do we measure quality in software?

7.5 Please characterize a software *module*.

7.6 What does an interface define?

7.7 What is the main benefit of the information hiding principle?

7.8 Why is it beneficial to separate code and data of a program?

7.9 Develop a number class *Fraction* that is able to exactly represent a fraction value. Ensure that the class implements the methods required by the abstract superclass *Number*.

```
import java.util.Iterator; import java.util.Vector;
public class Client {
  private static Vector<Number> v = new Vector<Number>();
  public Vector<Number> select(String opCode, double val) {
    Vector<Number> sel = new Vector<Number>();
    Number num;
    Iterator i = v.iterator();
    while(i.hasNext()) {
      num = (Number) i.next();
      if (opCode.equals("<") & (num.doubleValue() < val)) sel.add(num);
      if (opCode.equals("=") & (num.doubleValue() == val)) sel.add(num);
      if (opCode.equals(">") & (num.doubleValue() > val)) sel.add(num);
    }
    return sel;
  }
  public static void main(String[] args) {
    Client me = new Client();
    Number[] n = {1, 3.14, 1/3., 1f, 1.4, 4};
    for(int i=0; i < 6; i++) v.add(n[i]);
    String[] op = ">", "<";
    double[] w = 1, 1;
    System.out.println("Numbers in collection v with values ...");
    System.out.println("> 1 are: "+me.select(op[0],Double.valueOf(w[0])));
    System.out.println("< 1 are: "+me.select(op[1],Double.valueOf(w[1])));
  }
}
```

Figure 7.31 Application example of iterator pattern in Java

Hint: use integer variables for the nominator and denominator.

7.10 Develop a border decorator class *BorderDecorator* that provides an border for the *ColorDecorator* class. The stroke of the border should be adjustable. Hint: define *BorderDecorator* as a subclass of *ColorDecorator* and hold the stroke in a instance variable.

Bibliographical Notes

The computer scientist D. L. Parnas introduced the term *module* in a paper **?** that appeared in the Communications of the American Computer Association in 1972.

An other milestone in software construction was the book “Design Patterns” written by Gamma, Helm, Johnson, and Vlissides (**?**. This publication of design patterns should help to start a “cultural revolution” in the development of reusable code for typical software problems. The authors were therefore later called “gang of four” in analogy to the four chineese communists who started the cultural revolution in 1966.