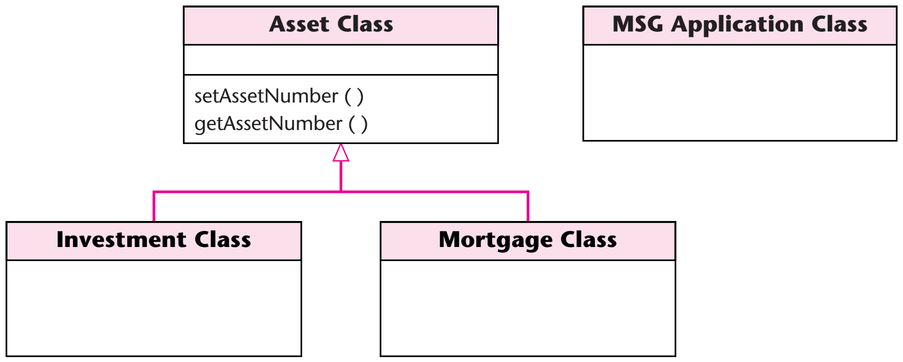
FIGURE 14.15 Part of the class diagram for the MSG Foundation case study with methods setAssetNumber and getAssetNumber assigned to **Asset Class**.



can be applied not only to instances of **Asset Class** but also, as a consequence of inheritance, to instances of every subclass of **Asset Class**, that is, to instances of **Investment Class** and **Mortgage Class**. Similarly, method getAssetNumber should also be allocated to the superclass **Asset Class**.

Assigning the other methods to the appropriate classes is equally straightforward. The resulting design is shown in Appendix G.

Step 2. Perform the Detailed Design

Next, the detailed design is built by taking each method and determining what it does. Figure 14.16 shows the detailed design (in a PDL for Java) of a method computeEstimatedFunds of class **EstimateFundsForWeek** of the MSG Foundation case study. This method invokes method totalWeeklyNetPayments of class **Mortgage** shown in Figure 14.17.

The steps of object-oriented design are summarized in How to Perform Box 14.3.

14.9 The Design Workflow

The overall aim of the **design workflow** is to refine the artifacts of the analysis workflow until the material is in a form that can be implemented by the programmers. The input to the design workflow is therefore the analysis workflow artifacts (Chapter 13). During the design workflow, these artifacts are iterated and incremented until they are in a format that can be utilized by the programmers.

How to Perform Object-Oriented Design Box 14.3

- Complete the class diagram.
- Perform the detailed design.

FIGURE 14.16 The detailed design of method computeEstimatedFunds of class **EstimateFundsForWeek** of the MSG Foundation case study.

```
public static void computeEstimatedFunds( )
This method computes the estimated funds available for the week.
{
  float expectedWeeklyInvestmentReturn;                 (expected weekly investment return)
  float expectedTotalWeeklyNetPayments = (float) 0.0;
                                                        (expected total mortgage payments
                                                        less total weekly grants)
  float estimatedFunds = (float) 0.0;                   (total estimated funds for week)
Create an instance of an investment record.
  Investment inv = new Investment ( );
Create an instance of a mortgage record.
  Mortgage mort = new Mortgage ( );
Invoke method totalWeeklyReturnOnInvestment.
  expectedWeeklyInvestmentReturn = inv.totalWeeklyReturnOnInvestment ( );
Invoke method expectedTotalWeeklyNetPayments            (see Figure 14.17)
  expectedTotalWeeklyNetPayments = mort.totalWeeklyNetPayments ( );
Now compute the estimated funds for the week.
  estimatedFunds = (expectedWeeklyInvestmentReturn
       – (MSGApplication.getAnnualOperatingExpenses ( ) / (float) 52.0)
       + expectedTotalWeeklyNetPayments);

Store this value in the appropriate location.
  MSGApplication.setEstimatedFundsForWeek (estimatedFunds);
} // computeEstimatedFunds
```

One aspect of this iteration and incrementation is the identification of methods and their allocation to the appropriate classes. Another aspect is performing the detailed design. These two steps constitute the object-oriented design component of the design workflow.

In addition to performing the object-oriented design, many decisions have to be made as part of the design workflow. One such decision is the selection of the programming language in which the software product will be implemented. This process is described in detail in Chapter 15. Another decision is how much of existing software products to reuse in the new software product to be developed. Reuse is described in Chapter 8. Portability is another important design decision; this topic, too, is described in Chapter 8. Also, large software products are often implemented on a network of computers; yet another design decision is the allocation of each software component to the hardware component on which it is to run.

The major motivation behind the development of the Unified Process was to present a methodology that could be used to develop large-scale software products, typically, 500,000 lines of code or more. On the other hand, the implementations of the MSG Foundation case study in Appendices H and I are less than 5000 lines of C++ and Java, respectively. In other

FIGURE 14.17 The detailed design of method totalWeekly-NetPayments of class **Mortgage** of the MSG Foundation case study.

```
public float totalWeeklyNetPayments ( )
This method computes the net total weekly payments made by the mortgagees, that is, the expected total weekly
mortgage amount less the expected total weekly grants.
{
  File mortgageFile = new File ("mortgage.dat");            (file of mortgage records)
  float expectedTotalWeeklyMortgages = (float) 0.0;         (expected total weekly mortgage payments)
  float expectedTotalWeeklyGrants = (float) 0.0;            (expected total weekly grants)
  float interestPayment;                                    (interest payment)
  float escrowPayment;                                      (escrow payment)
  float capitalRepayment;                                   (capital repayment)
  float weeklyPayment;                                      (mortgage payment for week)
  float maximumPermittedMortgagePayment;                    (maximum amount the couple may pay)
Open the file of mortgages, name it inFile, and read each element in turn.
{
  read (inFile);
Compute the interest payment, escrow payment, and capital repayment for this mortgage.
  interestPayment = mortgageBalance * INTEREST_RATE / WEEKS_IN_YEAR ;
  escrowPayment = (annualPropertyTax + annualInsurancePremium) / WEEKS_IN_YEAR;
  capitalRepayment = weeklyPrincipalAndInterestPayment – interestPayment;
  mortgageBalance –= capitalRepayment;
First assume that the couple can pay the mortgage in full, without a grant.
  weeklyPayment = weeklyPrincipalAndInterestPayment + escrowPayment;
Add the weekly Principal and Interest payment to the running total of mortgage payments
  expectedTotalWeeklyMortgages += weeklyPrincipalAndInterestPayment;
Now determine how much the couple can actually pay.
  maximumPermittedMortgagePayment = currentWeeklyIncome *
      MAXIMUM_PERC_OF_INCOME;
If a grant is needed, add the grant amount to the running total of grants
  if (weeklyPayment > maximumPermittedMortgagePayment)
  expectedTotalWeeklyGrants += weeklyPayment – maximumPermittedMortgagePayment;
  }
Close the file of mortgages. Return the total expected net payments for the week.
  return (expectedTotalWeeklyMortgages – expectedTotalWeeklyGrants);
} // totalWeeklyNetPayments
```

words, the Unified Process is intended primarily for software products at least 100 times larger than the MSG Foundation case study presented in this book. Accordingly, many aspects of the Unified Process are inapplicable to this case study. For instance, an important part of the analysis workflow is to partition the software product into analysis packages. Each **package** consists of a set of related classes, usually of relevance to a small subset of the actors, that can be implemented as a single unit. For example, accounts payable, accounts receivable, and general ledger are typical analysis packages. The concept underlying analysis packages is that it is much easier to develop smaller software products than larger software products. Accordingly, a large software product is easier to develop if it can be decomposed into relatively independent packages. Decomposing a software product into packages is an example of divide-and-conquer (Section 5.3).

This idea of decomposing a large workflow into relatively independent smaller workflows is carried forward to the design workflow. Here, the objective is to break up the upcoming implementation workflow into manageable pieces, termed **subsystems**. Again, it does not make sense to break up the MSG Foundation case study into subsystems; the case study is just too small.

There are two reasons why larger workflows are broken into subsystems:

1. As previously explained, it is easier to implement a number of smaller subsystems than one large system. That is, breaking up a software product into subsystems is another example of divide-and-conquer (Section 5.3).
2. If the subsystems to be implemented are indeed relatively independent, then they can be implemented by programming teams working in parallel. This results in the software product as a whole being delivered sooner.

Recall from Section 8.5.4 that the *architecture* of a software product includes the various components and how they fit together. The allocation of components to subsystems is a major part of the architectural task. Deciding on the architecture of a software product is by no means easy and, in all but the smallest software products, is performed by a specialist, the software **architect**.

In addition to being a technical expert, an architect needs to know how to make **trade-offs**. A software product has to satisfy the functional requirements, that is, the use cases. It also needs to satisfy the nonfunctional requirements, including portability (Chapter 8), reliability (Section 6.4.2), robustness (Section 6.4.3), maintainability, and security. But it needs to do all these things within budget and time constraints. It is almost never possible to develop a software product that satisfies all its requirements, both functional and nonfunctional, and finish the project within the cost and time constraints; compromises almost always have to be made. The client has to relax some of the requirements, increase the budget, or move the delivery deadline, or do more than one of these. The architect must assist the client's decision making by clearly mapping out the trade-offs.

In some cases the trade-offs are obvious. For example, the architect may point out that a set of security requirements that conform to a new high-security standard are going to take a further 3 months and $350,000 to incorporate in the software product. If the product is an international banking network, the issue is moot—there is no way that the client could possibly agree to compromise on security in any way. However, in other instances, the client needs to make critical determinations regarding trade-offs and has to rely on the technical expertise of the architect to assist in coming to the right business decision. For example, the architect might point out that deferring a particular requirement until the software product has been delivered and is being maintained may save $150,000 now but will cost $300,000 to incorporate later (see Figure 1.6). The decision whether or not to defer a requirement can be made only by the client, but he or she needs the technical expertise of the architect to assist in coming to the correct decision.

The architecture of a software product is a vital factor in the delivered product's success or a failure. And the critical decisions regarding the architecture have to be made while performing the design workflow. If the requirements workflow is badly performed, it is still possible to have a successful project, provided additional time and money are spent on the analysis workflow. Similarly, if the analysis workflow is inadequate, it is possible to recover by making an extra effort as part of the design workflow. But if the architecture is suboptimal, there is no way to recover; the architecture must immediately be redesigned. It is therefore essential that the development team include an architect with the necessary technical expertise and people skills.

14.10 The Test Workflow: Design

The goal of testing the design is to verify that the specifications have been accurately and completely incorporated into the design as well as to ensure the correctness of the design itself. For example, the design must have no logic faults, and all interfaces must be correctly defined. It is important that any faults in the design be detected before coding commences; otherwise, the cost of fixing the faults will be considerably higher, as reflected in Figure 1.6. Design faults can be detected by means of design inspections as well as design walkthroughs. Design inspections are discussed in the remainder of this section, but the remarks apply equally to design walkthroughs.

When the product is transaction oriented (Section 14.4), the design inspection should reflect this [Beizer, 1990]. Inspections that include all possible transaction types should be scheduled. The reviewer should relate each transaction in the design to the specifications, showing how the transaction arises from the specification document. For example, if the application is an automated teller machine, a transaction corresponds to each operation the customer can perform, such as deposit to or withdraw from a credit card account. In other instances, the correspondence between specifications and transactions is not necessarily one-to-one. In a traffic-light control system, for example, if an automobile driving over a sensor pad results in the system deciding to change a particular light from red to green in 15 seconds, then further impulses from that sensor pad may be ignored. Conversely, to speed traffic flow, a single impulse may cause a whole series of lights to be changed from red to green.

Restricting reviews to **transaction-driven inspections** does not detect cases where the designers have overlooked instances of transactions required by the specifications. To take an extreme example, the specifications for the traffic-light controller may stipulate that between 11:00 P.M. and 6:00 A.M. all lights are to flash yellow in one direction and red in the other direction. If the designers overlooked this stipulation, then clock-generated transactions at 11:00 P.M. and 6:00 A.M. would not be included in the design; and if these transactions were overlooked, they could not be tested in a design inspection based on transactions. Therefore, it is not adequate to schedule design inspections that are just transaction driven; specification-driven inspections also are essential to ensure that no statement in the specification document has been either overlooked or misinterpreted.

Case Study

14.11 The Test Workflow: The MSG Foundation Case Study

Now that the design is apparently complete, all aspects of the design of the MSG Foundation case study must be checked by means of a design inspection (Section 6.2.3). In particular, each design artifact must be examined. Even if no faults are found, it is possible that the design will change again, perhaps radically, when the MSG Foundation case study is implemented.

14.12 Formal Techniques for Detailed Design

One technique for detailed design has already been presented. In Section 5.1, a description of stepwise refinement was given. It then was applied to detailed design using flowcharts. In addition to stepwise refinement, formal techniques can be used to advantage in detailed design. Chapter 6 suggests that implementing a complete product and then proving it correct could be counterproductive. However, developing the proof and the detailed design in parallel and carefully testing the code as well is quite a different matter. Formal techniques applied to detailed design can greatly assist in three ways:

1. The state of the art in proving correctness is such that, although it generally cannot be applied to a product as a whole, it can be applied to module-sized pieces of a product.
2. Developing a proof together with the detailed design should lead to a design with fewer faults than if correctness proofs were not used.
3. If the same programmer is responsible for both the detailed design and the implementation, then that programmer will feel confident that the detailed design is correct. This positive attitude toward the design should lead to fewer faults in the code.

14.13 Real-Time Design Techniques

As explained in Section 6.4.4, **real-time software** is characterized by hard time constraints, that is, time constraints of such a nature that, if a constraint is not met, information is lost. In particular, each input must be processed before the next input arrives. An example of such a system is a computer-controlled nuclear reactor. Inputs such as the temperature of the core and the level of the water in the reactor chamber are continually being sent to the computer that reads the value of each input and performs the necessary processing before the next input arrives. Another example is a computer-controlled intensive care unit. There are two types of patient data: routine information such as heart rate, temperature, and blood pressure of each patient, and emergency information, when the system deduces that the condition of a patient has become critical. When such emergencies occur, the software must process both the routine inputs and the emergency-related inputs from one or more patients.

A characteristic of many real-time systems is that they are implemented on distributed hardware. For example, software controlling a fighter aircraft may be implemented on five computers: one to handle navigation, another the weapons system, a third for electronic countermeasures, a fourth to control the flight hardware such as wing flaps and engines, and the fifth to propose tactics in combat. Because hardware is not totally reliable, there may be additional backup computers that automatically replace a malfunctioning unit. Not only does the design of such a system have major communications implications, but timing issues, over and above those of the type just described, arise as a consequence of the distributed nature of the system. For example, under combat conditions, the tactical computer might suggest that the pilot should climb, whereas the weapons computer recommends that the pilot go into a dive so that a particular weapon may be launched under optimal conditions. However, the human pilot decides to move the stick to the right, thereby sending a signal to the flight hardware computer to make the necessary adjustments so that the plane banks in the indicated direction. All this information must be managed carefully in such a way that the actual motion of the plane takes precedence in every way over suggested maneuvers. Furthermore, the actual motion must be relayed to the tactical and weapons computers so that new suggestions can be formulated in the light of actual, rather than suggested, conditions.

A further difficulty with real-time systems is the problem of synchronization. Suppose that a real-time system is to be implemented on distributed hardware. Situations such as deadlock (or deadly embrace) can arise when two operations each have exclusive use of a data item and each requests exclusive use of the other's data item in addition. Of course, deadlock does not occur only in real-time systems, implemented on distributed hardware. But it is particularly troublesome in real-time systems where there is no control over the order or timing of the inputs, and the situation can be complicated by the distributed nature of the hardware. In addition to deadlock, other synchronization problems are possible, including race conditions; for details, the reader may refer to [Silberschatz, Galvin, and Gagne, 2002] or other operating systems textbooks.

From these examples it is clear that the major difficulty with regard to the design of real-time systems is ensuring that the timing constraints are met by the design. That is, the design technique should provide a mechanism for checking that, when implemented, the design is able to read and process incoming data at the required rate. Furthermore, it should be possible to show that synchronization issues in the design also have been addressed correctly.

Since the beginning of the computer age, advances in hardware technology have outstripped, in almost every respect, advances in software technology. Therefore, although the hardware exists to handle every aspect of the real-time systems described previously, software design technology has lagged behind considerably. In some areas of real-time software engineering, major progress has been made. For instance, many of the analysis techniques of Chapters 12 and 13 can be used to specify real-time systems. Unfortunately, software design has not yet reached the same level of sophistication. Great strides indeed are being made, but the state of the art is not yet comparable to what has been achieved with regard to analysis techniques. Because almost any design technique for real-time systems is preferable to no technique at all, a number of real-time design techniques are used in practice. But, there still is a long way to go before it will be possible to design real-time systems such as those described previously and be certain that, before the system has been implemented, every real-time constraint will be met and synchronization problems cannot arise.

Older real-time design techniques are extensions of non-real-time techniques to the real-time domain. For example, structured development for real-time systems (SDRTS) [Ward and Mellor, 1985] essentially is an extension of structured systems analysis (Section 12.3), data flow analysis (Section 14.3), and transaction analysis (Section 14.4) to real-time software. The development technique includes a component for real-time design. Newer techniques are described in [Liu, 2000] and [Gomaa, 2000].

As stated previously, it is unfortunate that the state of the art of real-time design is not as advanced as one would wish. Nevertheless, efforts are under way to improve the situation.

14.14 CASE Tools for Design

As stated in Section 14.10, a critical aspect of design is testing that the design artifacts accurately incorporate all aspects of the analysis. What is therefore needed is a CASE tool that can be used both for the analysis artifacts and the design artifacts, a so-called front-end or upperCASE tool (as opposed to a back-end or lowerCASE tool, which assists with the implementation artifacts).

A number of upperCASE tools are on the market. Some of the more popular ones include Analyst/Designer, Software through Pictures, and System Architect. UpperCASE tools generally are built around a data dictionary. The CASE tool can check that every field of every record in the dictionary is mentioned somewhere in the design or that every item in the design is reflected in the data flow diagram. In addition, many upperCASE tools incorporate a consistency checker that uses the data dictionary to determine that every item in the design has been declared in the specifications and conversely that every item in the specifications appears in the design.

Furthermore, many upperCASE tools incorporate screen and report generators. That is, the client can specify what items are to appear in a report or on an input screen and where and how each item is to appear. Because full details regarding every item are in the data dictionary, the CASE tool can easily generate the code for printing the report or displaying the input screen according to the client's wishes. Some upperCASE products also incorporate management tools for estimating and planning.

With regard to object-oriented design, Together, IBM Rational Rose, and Software through Pictures provide support for this workflow within the context of the complete object-oriented life cycle. Open-source CASE tools of this type include ArgoUML.

14.15 Metrics for Design

A variety of metrics can be used to describe aspects of the design. For example, the number of code artifacts (modules or classes) is a crude measure of the size of the target product. Cohesion and coupling are measures of the quality of the design, as are fault statistics. As with all other types of inspection, it is vital to keep a record of the number and type of design faults detected during a design inspection. This information is used during code inspections of the product and in design inspections of subsequent products.

The **cyclomatic complexity** *M* of a detailed design is the number of binary decisions (predicates) plus 1 [McCabe, 1976] or, equivalently, the number of branches in the code artifact. It has been suggested that cyclomatic complexity is a metric of design quality; the lower the value of *M*, the better. A strength of this metric is that it is easy to compute. However, it has an inherent problem. Cyclomatic complexity is purely a measure of the control complexity; the data complexity is ignored. That is, *M* does not measure the complexity of a code artifact that is data driven, such as by the values in a table. For example, suppose a designer is unaware of the C++ library function toascii and designs a code artifact from scratch that reads a character input by the user and returns the corresponding ASCII code (an integer between 0 and 127). One way of designing this is by means of a 128-way branch implemented by means of a **switch** statement. A second way is to have an array containing the 128 characters in ASCII code order and utilize a loop to compare the character input by the user with each element of the array of characters; the loop is exited when a match is obtained. The current value of the loop variable then is the corresponding ASCII code. The two designs are equivalent in functionality but have cyclomatic complexities of 128 and 1, respectively.

When the classical paradigm is used, a related class of metrics for the design phase is based on representing the architectural design as a directed graph with the modules represented by nodes and the flows between modules (procedure and function calls) represented by arcs. The **fan-in** of a module can be defined as the number of flows into the module plus the number of global data structures accessed by the module. The **fan-out** similarly is the number of flows out of the module plus the number of global data structures updated by the module. A measure of complexity of the module then is given by $\textit{length} \times (\textit{fan-in} \times \textit{fan-out})^2$ [Henry and Kafura, 1981], where **length** is a measure of the size of the module (Section 9.2.1). Because the definitions of *fan-in* and *fan-out* incorporate global data, this metric has a data-dependent component. Nevertheless, experiments have shown that this metric is no better a measure of complexity than simpler metrics, such as cyclomatic complexity [Kitchenham, Pickard, and Linkman, 1990; Shepperd, 1990].

The issue of design metrics is complicated even more when the object-oriented paradigm is used. For example, the cyclomatic complexity of a class usually is low, because many classes typically include a large number of small, straightforward methods. Furthermore, as previously pointed out, cyclomatic complexity ignores data complexity. Because data and operations are equal partners within the object-oriented paradigm, cyclomatic complexity overlooks a major component that could contribute to the complexity of an object. Therefore, metrics for classes that incorporate cyclomatic complexity generally are of little use.

A number of object-oriented design metrics have been put forward, for example, in [Chidamber and Kemerer, 1994]. These and other metrics have been questioned on both theoretical and experimental grounds [Binkley and Schach, 1996; 1997; 1998].

14.16 Challenges of the Design Workflow

As pointed out in Sections 12.16 and 13.22, it is important not to do too much in the analysis workflow; that is, the analysis team must not prematurely start parts of the design workflow. In the design workflow, the design team can go wrong in two ways: by doing too much and by doing too little.

Consider the PDL (pseudocode) detailed design of Figure 14.7. The temptation is strong for a designer who enjoys programming to write the detailed design in C++ or Java, rather than PDL. That is, instead of sketching the detailed design in pseudocode, the designer may all but code the class. This takes longer to write than just outlining the class and longer to fix if a fault is detected in the design (see Figure 1.6). Like the analysis team, the members of the design team must firmly resist the urge to do more than what is required of them.

At the same time, the design team must be careful not to do too little. Consider the tabular detailed design of Figure 14.6. If the design team is in a hurry, it may decide to shrink the detailed design to just the narrative box. The team may even decide that the programmers should do the detailed design by themselves. Either of these decisions would be a mistake. A primary reason for the detailed design is to ensure that all interfaces are correct. The narrative box by itself is inadequate for this purpose; no detailed design at all clearly is even less helpful. Therefore, one challenge of the design workflow is for the designers to do just the correct amount of work.

In addition, there is a much more significant challenge. In "No Silver Bullet" (see Just in Case You Wanted to Know Box 3.4), Brooks [1986] decries the lack of what he terms *great designers*, that is, designers who are significantly more outstanding than the other members of the design team. In Brooks's opinion, the success of a software project depends critically on whether the design team is led by a great designer. Good design can be taught; great design is produced only by great designers, and they are "very rare."

The challenge, then, is to grow great designers. They should be identified as early as possible (the best designers are not necessarily the most experienced), assigned a mentor, provided a formal education as well as apprenticeships to great designers, and allowed to interact with other designers. A specific career path should be available for these designers, and the rewards they receive should be commensurate with the contribution that only a great designer can make to a software development project.

Chapter Review

The design workflow is introduced in Section 14.1. There are three basic approaches to design: operation-oriented design (Section 14.2), data-oriented design (Section 14.5), and object-oriented design (Section 14.6). Two instances of operation-oriented design are described, data flow analysis (Section 14.3) and transaction analysis (Section 14.4). Object-oriented design is applied to the elevator problem case study in Section 14.7 and to the MSG Foundation case study in Section 14.8. The design workflow is presented in Section 14.9. The design aspects of the test workflow are described in Section 14.10 and applied to the MSG Foundation case study in Section 14.11. Formal techniques for detailed design are discussed in Section 14.12. Real-time system design is described in Section 14.13. CASE tools and metrics for the design workflow are presented in Sections 14.14 and 14.15, respectively. The chapter concludes with a discussion of the challenges of the design workflow (Section 14.16).

An overview of the MSG Foundation case study for Chapter 14 appears in Figure 14.18, and for the elevator problem in Figure 14.19.

FIGURE 14.18 Overview of the MSG Foundation case study for Chapter 14.

Object-oriented design	Section 14.8
Overall class diagram	Figure 14.13
Part of overall class diagram with attribute formats added	Figure 14.14
Detailed design	Appendix G

FIGURE 14.19 Overview of the elevator problem case study for Chapter 14.

Object-oriented design	Section 14.7
Detailed class diagram	Figure 14.11

For Further Reading

Data flow analysis and transaction analysis are described in books such as [Gane and Sarsen, 1979] and [Yourdon and Constantine, 1979].

The March–April 2005 issue of *IEEE Software* contains a number of papers on design. Designing for recovery, that is, designing software to detect, react, and recover from exceptional conditions, is described in [Wirfs-Brock, 2006].

Briand, Bunse, and Daly [2001] discuss the maintainability of object-oriented designs. A comparison of both object-oriented and classical design techniques appears in [Fichman and Kemerer, 1992]. The redesign of an air traffic control system is described in [Jackson and Chapin, 2000]. Design techniques for high-performance, reliable systems are given in [Stolper, 1999]. A probabilistic approach to estimating the change proneness of an object-oriented design appears in [Tsantalis, Chatzigeorgiou, and Stephanides, 2005]. A discussion as to whether object-oriented design is intuitive appears in [Hadar and Leron, 2008].

Formal design techniques are described in [Hoare, 1987]. The vital role played by the architect is described in [McBride, 2007]. Analogously to pair programming, pair design and its effectiveness are described in [Lui, Chan, and Nosek, 2008].

With regard to reviews during the design process, the original paper on design inspections is [Fagan, 1976]; detailed information can be obtained from that paper. Later advances in review techniques are described in [Fagan, 1986]. Architecture reviews are discussed in [Maranzano et al., 2005].

With regard to real-time design, specific techniques are to be found in [Liu, 2000] and [Gomaa, 2000]. A comparison of four real-time design techniques is found in [Kelly and Sherif, 1992]. A documentation-driven approach to the design of complex real-time systems is described in [Luqi, Zhang, Berzins, and Qiao, 2004]. The design of concurrent systems is described in [Magee and Kramer, 1999].

Metrics for design are described in [Henry and Kafura, 1981] and [Zage and Zage, 1993]. Metrics for object-oriented design are discussed in [Chidamber and Kemerer, 1994] and in [Binkley and Schach, 1996]. A model for object-oriented quality is presented in [Bansiya and Davis, 2002].

The proceedings of the International Workshops on Software Specification and Design are a comprehensive source for information on design techniques.

Key Terms

abstract data type design *476*
accessor *482*
architect *486*
architectural design *466*
class diagram *476*
cyclomatic complexity *491*
data flow analysis (DFA) *467*
data-oriented design *465*
design workflow *483*
detailed design *466*
fan-in *491*
fan-out *491*
general design *466*
high-level design *466*
length *491*
logical design *466*
low-level design *466*
modular design *466*
mutator *482*
object-oriented design (OOD) *476*
operation-oriented design *465*
package *486*
physical design *466*

point of highest abstraction of input *467*
point of highest abstraction of output *467*
program description language (PDL) *471*
pseudocode *471*
real-time software *488*
responsibility-driven design *477*
subsystem *486*
trade-off *486*
transaction *473*
transaction analysis *475*
transaction-driven inspections *487*

Problems

14.1 Starting with your DFD for Problem 12.9, use data flow analysis to design a product for determining whether a bank statement is correct.

14.2 Use transaction analysis to design the software to control an ATM (Problem 8.9). At this stage omit error-handling capabilities.

14.3 Now take your design for Problem 14.2 and add modules to perform error handling. Carefully examine the resulting design and determine the cohesion and coupling of the modules. Be on the lookout for situations such as that depicted in Figure 14.10.

14.4 Two different techniques for depicting a detailed design are presented in Section 14.3.1 (Figures 14.6 and 14.7). Compare and contrast the two techniques.

14.5 Starting with your data flow diagram for the automated library circulation system (Problem 12.11), design the circulation system using data flow analysis.

14.6 Repeat Problem 14.5 using transaction analysis. Which of the two techniques did you find to be more appropriate?

14.7 Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Elevator Subcontroller Class**.

14.8 Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Floor Subcontroller Class**.

14.9 Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Sensor Class**.

14.10 Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Floor Button Class**.

14.11 Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Elevator Button Class**.

14.12 Complete the detailed class diagram for the elevator problem case study (Figure 14.11) by listing the methods of the form Send message to **C Class** . . . that need to be included in the **Scheduler Class**.

14.13 (Analysis and Design Project) Starting with your object-oriented analysis for the automated library circulation system (Problem 13.19), design the library system using object-oriented design.

14.14 (Analysis and Design Project) Starting with your object-oriented analysis for the product for determining whether a bank statement is correct (Problem 13.20), design the software using object-oriented design.

14.15 (Analysis and Design Project) Starting with your object-oriented analysis for the ATM software (Problem 13.21), design the ATM software using object-oriented design.

14.16 (Term Project) Starting with your specifications of Problem 12.20 or 13.22, design the Chocoholics Anonymous product (Appendix A). Use the design technique specified by your instructor.

14.17 (Case Study) Redesign the MSG Foundation product using data flow analysis.

14.18 (Case Study) Redesign the MSG Foundation product using transaction analysis.

14.19 (Case Study) The detailed design of Figures 14.16 and 14.17 is represented in PDL form. Represent the design using a tabular format. Which representation is superior? Give reasons for your answer.

14.20 (Readings in Software Engineering) Your instructor will distribute copies of [Hadar and Leron, 2008]. To what extent do you think that object-oriented design is intuitive?

References

[Bansiya and Davis, 2002] J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering* **28** (January 2002), pp. 4–17.

[Beizer, 1990] B. Beizer, *Software Testing Techniques,* 2nd ed., Van Nostrand Reinhold, New York, 1990.

[Binkley and Schach, 1996] A. B. Binkley and S. R. Schach, "A Comparison of Sixteen Quality Metrics for Object-Oriented Design," *Information Processing Letters* **57** (No. 6, June 1996), pp. 271–75.

[Binkley and Schach, 1997] A. B. Binkley and S. R. Schach, "Toward a Unified Approach to Object-Oriented Coupling," *Proceedings of the 35th Annual ACM Southeast Conference*, Murfreesboro, TN, April 2-4, 1997, IEEE, pp. 91–97.

[Binkley and Schach, 1998] A. B. Binkley and S. R. Schach, "Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures," *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan, April 1988, IEEE, pp. 542–55.

[Briand, Bunse, and Daly, 2001] L. C. Briand, C. Bunse, and J. W. Daly, "A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs," *IEEE Transactions on Software Engineering* **27** (June 2001), pp. 513–30.

[Brooks, 1986] F. P. Brooks, Jr., "No Silver Bullet," in: *Information Processing '86*, H.-J. Kugler (Editor), Elsevier North-Holland, New York, 1986; reprinted in: *IEEE Computer* **20** (April 1987), pp. 10–19.

[Chidamber and Kemerer, 1994] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering* **20** (June 1994), pp. 476–93.

[Fagan, 1976] M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal* **15** (No. 3, 1976), pp. 182–211.

[Fagan, 1986] M. E. Fagan, "Advances in Software Inspections," *IEEE Transactions on Software Engineering* **SE-12** (July 1986), pp. 744–51.

[Fichman and Kemerer, 1992] R. G. Fichman and C. F. Kemerer, "Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique," *IEEE Computer* **25** (October 1992), pp. 22–39.

[Flanagan, 2005] D. Flanagan, *Java in a Nutshell: A Desktop Quick Reference*, 5th ed., O'Reilly and Associates, Sebastopol, CA, 2005.

[Gane and Sarsen, 1979] C. Gane and T. Sarsen, *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, Englewood Cliffs, NJ, 1979.

[Goldberg and Robson, 1989] A. Goldberg and D. Robson, *Smalltalk-80: The Language,* Addison-Wesley, Reading, MA, 1989.

[Gomaa, 2000] H. GOMAA, *Designing Concurrent, Distributed, and Real-time Applications with UML*, Addison-Wesley, Reading, MA, 2000.

[Hadar and Leron, 2008] "How Intuitive Is Object-Oriented Design?" *Communications of the ACM* **51** (May 2008), pp. 41–46.

[Henry and Kafura, 1981] S. M. HENRY AND D. KAFURA, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering* **SE-7** (September 1981), pp. 510–18.

[Hoare, 1987] C. A. R. HOARE, "An Overview of Some Formal Methods for Program Design," *IEEE Computer* **20** (September 1987), pp. 85–91.

[ISO/IEC 8652, 1995] *Programming Language Ada: Language and Standard Libraries*, ISO/IEC 8652, International Organization for Standardization, International Electrotechnical Commission, Geneva, Switzerland, 1995.

[Jackson, 1975] M. A. JACKSON, *Principles of Program Design*, Academic Press, New York, 1975.

[Jackson and Chapin, 2000] D. JACKSON AND J. CHAPIN, "Redesigning Air Traffic Control: An Exercise in Software Design," *IEEE Software* **17** (May–June 2000), pp. 63–70.

[Kelly and Sherif, 1992] J. C. KELLY AND J. S. SHERIF, "A Comparison of Four Design Methods for Real-Time Software Development," *Information and Software Technology* **34** (February 1992), pp. 74–82.

[Kitchenham, Pickard, and Linkman, 1990] B. A. KITCHENHAM, L. M. PICKARD, AND S. J. LINKMAN, "An Evaluation of Some Design Metrics," *Software Engineering Journal* **5** (January 1990), pp. 50–58.

[Liu, 2000] J. W. S. LIU, *Real Time Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[Lui, Chan, and Nosek, 2008] K. M. LUI, K. C. C. CHAN, AND J. T. NOSEK, "The Effect of Pairs in Program Design Tasks," *IEEE Transactions on Software Engineering* **34** (March–April 2008), pp. 197–211.

[Luqi, Zhang, Berzins, and Qiao, 2004] LUQI, L. ZHANG, V. BERZINS, AND Y. QIAO, "Documentation Driven Development for Complex Real-Time Systems," *IEEE Transactions on Software Engineering* **30** (December 2004), pp. 936–52.

[Magee and Kramer, 1999] J. MAGEE AND J. KRAMER, *Concurrency: State Models & Java Programs*, John Wiley and Sons, New York, 1999.

[Maranzano et al., 2005] J. F. MARANZANO, S. A. ROZSYPAL, G. H. ZIMMERMAN, G. W. WARNKEN, P. E. WIRTH, AND D. M. WEISS, "Architecture Reviews: Practice and Experience," *IEEE Software* **22** (March–April 2005), pp. 34–43.

[McCabe, 1976] T. J. MCCABE, "A Complexity Measure," *IEEE Transactions on Software Engineering* **SE-2** (December 1976), pp. 308–20.

[McBride, 2007] M. R. MCBRIDE, "The Software Architect," *Communications of the ACM* **50** (May 2007), pp. 75–81.

[Orr, 1981] K. ORR, *Structured Requirements Definition*, Ken Orr and Associates, Topeka, KS, 1981.

[Shepperd, 1990] M. SHEPPERD, "Design Metrics: An Empirical Analysis," *Software Engineering Journal* **5** (January 1990), pp. 3–10.

[Silberschatz, Galvin, and Gagne, 2002] A. SILBERSCHATZ, P. B. GALVIN, AND G. GAGNE, *Operating System Concepts,* 6th ed., Addison-Wesley, Reading, MA, 2002.

[Stolper, 1999] S. A. STOLPER, "Streamlined Design Approach Lands Mars Pathfinder," *IEEE Software* **16** (September–October 1999), pp. 52–62.

[Stroustrup, 2003] B. STROUSTRUP, *The C++ Standard: Incorporating Technical Corrigendum No. 1*, 2nd ed., John Wiley and Sons, New York, 2003.

[Tsantalis, Chatzigeorgiou, and Stephanides, 2005] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the Probability of Change in Object-Oriented Systems," *IEEE Transactions on Software Engineering* **31** (July 2005), pp. 601–14.

[Ward and Mellor, 1985] P. T. Ward and S. Mellor, *Structured Development for Real-Time Systems,* Vols. 1, 2, and 3, Yourdon Press, New York, 1985.

[Warnier, 1976] J. D. Warnier, *Logical Construction of Programs*, Van Nostrand Reinhold, New York, 1976.

[Wirfs-Brock, 2006] R. Wirfs-Brock, "Designing for Recovery," *IEEE Software* **23** (July–August 2006), pp. 11–13.

[Yourdon and Constantine, 1979] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice Hall, Englewood Cliffs, NJ, 1979.

[Zage and Zage, 1993] W. M. Zage and D. M. Zage, "Evaluating Design Metrics on Large-Scale Software," *IEEE Software* **10** (July 1993), pp. 75–81.