18 Chapter

Implementation Considerations

Key Takeaway Points

- Everyone in the team should follow the same coding standards.
- Test-driven development, pair programming, and code review improve the quality of the code.
- Classes should be implemented according to their dependencies to reduce the need for test stubs.

Previous chapters presented object-oriented analysis and design. The design activity produces the sequence diagrams, state diagrams, activity diagrams, and the design class diagram (DCD). Moreover, the classes are logically organized into packages. In this chapter, the design is converted into source code, which is compiled into executable programs. During this process, the team must consider several factors that affect the implementation. These include coding standards, how to organize the classes and web pages, how to translate the design into object-oriented programs, and how to assign the implementation work to the team members. Throughout this chapter, you will learn the following:

- What are coding standards, and how do you implement them in your teamwork?
- How do you organize the source files and executable programs?
- How do you convert the design diagrams to code skeletons?
- How do you assign the implementation work to team members?
- How do you conduct test-driven development, pair programming, and ping-pong programming?

18.1 CODING STANDARDS

Defining and complying to coding standards are an important aspect of implementation. Experiences show that poorly written code requires rework, which increases development and maintenance costs. In the worst case, the code has to be abandoned.

On the other hand, a good practice of coding standards improves software productivity and quality while reducing cost and time to market. Therefore, most companies develop and require programmers to comply with company-specific coding standards. Good practices of coding standards require an understanding of the following, which are the focuses of this section:

1. What are coding standards?
2. Why use coding standards?
3. What are the guidelines for practicing coding standards?

18.1.1 What Are Coding Standards?

Generally speaking, coding standards are a set of rules that serve as requirements and guidelines for writing programs in an organization, or for a project. More specifically, coding standards should address the following:

1. Define the required and optional items of the coding standards.
2. Define the format and language used to specify the required and optional items.
3. Define the coding requirements and conventions.
4. Define the rules and responsibilities to create and implement the coding standards, as well as review and improve the practice.

The coding standards should define the required and optional items so that the programmers will know what to include. More importantly, if all projects use the same standards, then integration, collaboration, and software reuse are much easier to accomplish. Another advantage is that the list forms the basis for future improvement. Coding standards usually include the following major items:

1. *File header.* Many companies' coding standards require a file header at the beginning of each program file. It specifies the file location, version number, author, project, update history, and other useful information. Figure 18.1 provides a summary of file header items.
2. *Description of classes.* Many companies also require a functional description for each class. It includes:
   a. Purpose—a statement of the purpose of the class.
   b. Description of methods—a brief description of the purpose of each method, the parameters, return type, and other input and output such as files, database tables, and text fields accessed and updated by the method. The list should not include the ordinary get and set functions.
   c. Description of fields—a brief description of each field including the name of the field, data type, range of values, initialization requirement, and values of significance.
   d. In-code comments—comments that are inserted at places of the code to facilitate understanding and tool processing.

Figure 18.2 displays the file header from a real-world project and Figure 18.3 shows a part of the description of a class.

Item	Description
File name	full path name of the file that contains the program including the name of the hard drive
Version number	the version number and release number (useful for versioning, maintenance, and software reuse)
Author name	name of the programmer who creates the program file and writes the first version of the program
Project name	full name of the project for which the program is initially written
Organization	full name of the organization
Copyright	copyright information
Related Requirements	a list of requirements implemented by the program
List of classes*	a list of the names of the classes contained in the program file
Related documents	a list of related documents, and their URLs if available
Update history	a list of updates, each specifies the date of the update, who performs the update, what is updated, why the update, and possibly the impact, risks, and resolution measurements
Reviewers*	a list of reviewers and what each of them reviews
Test cases*	a list of test scripts along with their URLs if available
Functional Description	an overall description of the functionality and behavior of the program—that is, what the program does and how it interacts with its client.
Error messages	a list of error messages that the program may generate along with a brief description of each of them
Assumptions	a list of conditions that must be satisfied, or may affect the operation of the program
Constraints	a list of restrictions on the use of the program including restrictions on the input, environment, and various other variables

*optional item

FIGURE 18.1 Coding standards file header items

The coding standards should specify the language and format that should be used to describe the items. For example, the sentences used to specify the items should be in third person and simple present tense. Figure 18.2 illustrates an example format. The coding standards should specify the coding requirements and conventions. The difference is that requirements are compulsory and conventions leave the decision to the programmer. The organization and the project manager decide which rules are coding requirements and which are coding conventions. For convenience, no distinction is made between the two in the following presentation. Coding conventions include:

1. *Naming conventions.* These conventions specify rules for naming packages, modules, paths, files, classes, attributes, functions, constants, and the like. Naming conventions should help program understanding and maintenance. The names that are selected for the classes, attributes, functions, and variables should convey their functionality and semantics, and facilitate understanding of the program. Moreover, the names should be the same, or easily relate to the names in the design. Meaningless names such as “flag,” “temp,” “x,” “y,” “z,” and the like, must not be used. In addition, names must not contain space, control characters, or special letters.

```
/**
*  File Name : Rectangle.java
*
*  The University of Texas at Arlington
*  Software Engineering Center for Telecommunications
*  Object Oriented Testing Project
*
*  (c) Copyright 1998 University of Texas at Arlington
*  ALL RIGHTS RESERVED
*
*  Input                       :   This class is constructed from input passed by the Layout
                                   class. This class is constructed by invoking the super class
                                   constructor. Refer to Shape.java for information on Input.
*  Output                      :   A Rectangle class to represent sequential statements is
                                   constructed and is used to draw a rectangle of the Flowgraph.
*  Supported Requirements      :   BBD_R19, BBD-R20, BBD-R23, BBD-R25, BBD-R26.
*  Classes in this file        :   Rectangle
*  Related Documents           :   BBD Analysis & Design Document
*                                  (GUI, Layout & Display)
*  Update History:
*
*  Date         Author                          Changes
*  ---------------------------------------------------------
*  6/20/98      Withheld                        Original
*
*
*  Functional Description      :   This file implements the Rectangle class of BBD Layout &
                                   Display Module. It is a subclass of Shape class. It generates
                                   the coordinates required for drawing a rectangle using AWT and
                                   stores them in Attributes[]. It also implements the draw
                                   method that draws the rectangle on the screen, inserts the
                                   associated text into it and places the node number by its side.
*  Error Messages              :
*  Constraints                 :   The text associated with the Shape is printed in the Shape
                                   only when the zoomRatio is above 3.
*  Assumptions                 :   It is assumed that the text will not be visible when the
                                   zoomRatio is less than 4 and hence it is not displayed.
*
**/
```

FIGURE 18.2 File header and functional description of a class

In Java, classes are stored in .java files with the same name as the class except inner classes. Class names have a capital initial. If the class name consists of several words, then each word has a capital initial and the words are lumped together like UndergraduateStudent. Attribute names, function names, and local variable names are like class names except that the first letter is not capitalized. Names for constants are all capital, like MAX.

2. *Formatting conventions.* Formatting conventions specify formatting rules used to arrange program statements. These include line break, indentation, alignment, and spacing. Most integrated development environments (IDE) define a default format, which can be changed according to the needs of the software development organization. In Figure 18.3, the formatting conventions are illustrated.
3. *In-code comment conventions.* If it is written properly, in-code comments facilitate program understanding and maintenance. Thus, the coding standards should

```
package bbd.display;
import java.io.*;
import java.awt.*;


/**
*
*  Purpose              :      This class is used for drawing a Rectangle Shape of the FlowGraph.
*
*  Usage Instructions   :
*
*  @author              Withheld
*  @Version             Original
*  @see                 java.awt.Graphics
*  @see                 http://java.sun.com/products/jdk/1.1/docs/api/
*                       java.awt.Graphics.html#_top_
**/

public class Rectangle extends Shape
{

/**
*
*  Method Name          :       Rectangle
*  Purpose              :       Constructor of the Rectangle class. It calls the super class
                                constructor to initialize the class variables.
*
**/

   public Rectangle( String id, int Type, String shape, int number,
                     String code )
   {

      super( id, Type, shape, number, code );

   }

/**
*
*  Method Name          :      setCoordinates
*  Purpose              :      It finalizes the coordinates of the Rectangle class at a zoomRatio
                               = 5 and stores them in attributes.
*  Miscellaneous        :      The attributes are initialized as follows. attributes[0] stores the
                               x coordinate of the top left corner. attributes[1] stores the y
                               coordinate of the top left corner. attribute[2] stores the width,
                               whereas attribute[3] stores the height of the Rectangle.

*
**/

   public void setCoordinates( )
   {

      System.out.println( "setCoordinates for Rectangle called" );
      x = GRAPH_START_X + ( x * XGAP );
      y = GRAPH_START_Y + ( y * YGAP );

      attributes[0] = x - WIDTH/2;
      attributes[1] = y - HEIGHT/2;
      attributes[2] = WIDTH;
      attributes[3] = HEIGHT;

   }
...
```

FIGURE 18.3 Description of a class

define the requirements and guidelines including the locations where in-code comments must, or should be provided, and the formats to write the comments so that tools can process them.

Finally, rules and responsibilities for creating and implementing the coding standards as well as reviewing and improving the practice must be defined. For example, should the standards be applied to all the programs created, or just to some of the programs created? Should the coding standards be applied to test case scripts, or test case programs? Should the test case programs use the same coding standards? Who should be responsible for defining the coding standards, and who should be responsible for educating the programmers to follow, or comply with the standards? Answers to these questions require a definition of rules and responsibilities.

18.1.2 Why Coding Standards?

Coding standards are not just a set of documentation rules. Good practices of coding standards bring about a number of benefits. First, the standards define a common set of rules for writing programs in the organization or project. Without the standards, the teams and team members would not include the items or write whatever they think should be included. In this case, there is no consistency between the program files produced by different teams and team members. The coding standards ensure the consistency of the program files produced by the teams and team members. This greatly facilitates project management, software integration, software reuse, configuration management, software maintenance, and personnel transfer. Second, the coding standards make it easy to understand the programs because all program files describe the same items using the same format. Moreover, the standards ensure that important items are included and properly specified to reap the benefits. Third, the coding standards help code review and test case generation because the functional description and in-code comments help the reviewers and testers understand the program. Finally, the standards facilitate the use of tools such as JavaDoc and static analysis tools. JavaDoc generates html pages as online documentation for the classes. Static analysis tools check the code and in-code document to detect potential problems and violation of the coding standards.

18.1.3 Code Review Checklist

Besides the coding standards discussed above, program inspection and code review also check for other items. The following are some of the items to be checked when reviewing the implementation of a class, component, or subsystem:

1. Does the program correctly implement the functionality and conform to the design specification?
2. Do the implemented class interfaces conform to the interfaces specified in the design class diagram?
3. Does the implementation comply to the coding standards?
4. Compute a number of required quality metrics and identify those that are worse than the required indicators. A metrics calculation tool is very useful in this regard.

5. Are programming constructs used correctly and properly? For example, improper uses include using an array to store a large number of elements, loops without proper termination conditions, and uncontrolled update to shared variables.
6. Does the program correctly implement the algorithms and data structures?
7. Are there any errors or anomalies in the definition and use of variables such as memory is allocated but not released, variables are defined but not used, or variables are incorrectly initialized?
8. Are there any incorrect uses of logical, arithmetic, or relational operators, incorrect invocation of functions, incorrect interfacing to devices, or incorrect handling of device interrupts?
9. Are there any potential performance bottlenecks, or an inability to fulfill timing constraints?

18.1.4 Guidelines for Practicing Coding Standards

Like everything else, coding standards could be overdone, resulting in standards that require everything to be documented in great detail. This is counterproductive because a lot of time and effort are wasted in describing things that are not important or obvious. Another problem is that the standards are defined but never implemented, checked, or enforced. The following guidelines are for practicing coding standards.

GUIDELINE 18.1 Define barely enough coding standards.

Coding standards should include only the most needed items. Barely enough coding standards are enough to reap the benefits while keeping the effort to a minimum. Barely enough coding standards should be determined by the stakeholders including the developers. This is because different domains, organizations, and projects should have different coding standards. For example, if the developers are familiar with the application domain, then the coding standards should require less. If the application is new, or difficult to understand for the developers, then the coding standards should require more descriptive information to be included in the program file.

GUIDELINE 18.2 The coding standards should be easy to understand and practice.

To reap the benefits of coding standards, developer support is essential. That is, the developers follow the coding standards when they write programs. This means that the standards should be easy to understand and practice; otherwise, the developers will give up sooner or later. Therefore, it is important to involve the developers. Moreover, the drafts of the coding standards should be reviewed and practiced by the developers, and modified according to feedback.

GUIDELINE 18.3 The coding standards should be documented and should include examples.

The first releasable version of the coding standards should be published as an official document. It should include examples to illustrate how to describe the required and optional items. This does not mean that the document should be a lengthy one. On the contrary, it should focus on what to do and how to do it.

GUIDELINE 18.4 Training on how to use the coding standards is helpful.

The training may require a couple of hours, but it is worthwhile because it helps the developers understand what is required. Moreover, the developers have the opportunity to ask questions to clear doubts.

GUIDELINE 18.5 The coding standards, once defined and published, should be practiced, enforced, and checked for compliance regularly.

If the coding standards are defined but not practiced, then they are useless. Therefore, it is important to put the standards into practice and check for their compliance regularly. Since the developers know that there will be a compliance check, they tend to comply. As time goes by, complying to the standards would become a part of the culture in the organization.

GUIDELINE 18.6 It is important to assign the responsibilities to individuals and make sure that all involved know the assignment.

Putting the coding standards to work requires clear assignment of responsibilities to individuals, that is, who is responsible for doing what. Moreover, the assignment should be announced at appropriate meetings to ensure that everybody knows his responsibilities. For example, the programmer is responsible for writing the file header, functional description, and the like. The reviewers are responsible for checking that the coding standards have been complied with. The configuration management personnel is responsible for making sure that all program files are properly reviewed and issues are addressed. These could be that the reviewers electronically sign their review reports indicating that issues are addressed as expected. Finally, it is important to establish checking mechanisms to ensure that everybody fulfills the responsibilities.

GUIDELINE 18.7 The practice of coding standards should involve stakeholders.

The success of standards practice depends on the support and cooperation of all stakeholders who are affected including the developers. Therefore, their opinions should be taken into consideration seriously by the management. It is worthwhile to spend time to communicate the rational, discuss issues, and resolve the issues.

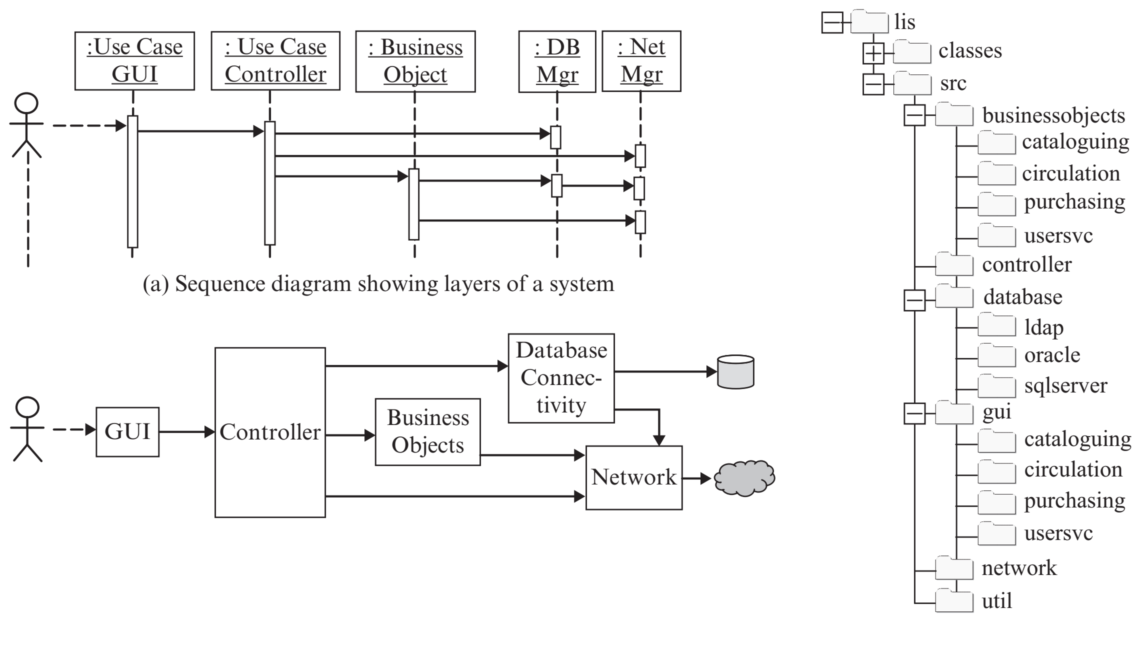
18.2 ORGANIZING THE IMPLEMENTATION ARTIFACTS

One important decision concerning implementation is deciding on the directory structures to store the source code, binary code, bytecode, web pages, and test cases. It is important because the directory structure closely resembles the system architecture, which is defined in the early stage of the project (see Chapter 6: Architectural Design).

The software architecture suggests the logical organization of the software artifacts, that is, grouping of the software artifacts into packages using UML package diagrams. Chapter 11 presented three approaches to organize the classes:

1. *Architectural-style organization.* This approach organizes the classes according to the building blocks of the software architecture. For example, Figure 11.12 shows the correspondence between the N-tier architecture and the packages for the library information system discussed in Chapter 11.
2. *Functional subsystem organization.* This approach organizes the classes according to the functional subsystems of the software system. Figure 11.11 shows the correspondence for the library information system.
3. *Hybrid organization.* This approach combines the architectural-style organization and the functional subsystem organization. Two approaches exist: the architectural-style functional subsystem organization and the functional subsystem architectural-style organization. Using the former, the classes in each of the architectural packages are organized according to the functional subsystems of the software system. This would organize the classes in the GUI package in Figure 11.12 into several subpackages of the GUI package, including gui.circulation, gui.cataloguing, gui.purchasing, gui.usersvc, and gui.interlibraryloan. The functional subsystem architectural-style organization does the opposite.

The package structure is useful for deriving the directory structure used to store the source files and executable programs. For example, an interactive system typically has sequence diagrams like the one in Figure 18.4(*a*), where the Use Case GUI and Use Case Controller objects represent use case-specific GUI and controller. The GUI



(a) Sequence diagram showing layers of a system

(b) Corresponding N-tier architecture

(c) Directory structure

FIGURE 18.4 N-tier architecture and directory structure

includes window-based, web-based, and other types of user interfaces. The Business Object represents one or more business objects that participate in the business process of the use case. As discussed in Chapter 17, the DBMgr is a part of a persistence framework that includes a DB Proxy and DB commands to access a database, which may locate at a remote site. The Net Mgr is responsible for handling network communication and marshaling. It is a part of a network communication framework. The layers of an N-tier architecture are in one-to-one correspondence to the layers in the sequence diagram, as illustrated in Figure 18.4(*b*). The layers represent subsystems or components and are implemented as modules or packages. For example, the database layer includes the DBMgr class, the proxy classes, and the database access command classes. These classes belong to the database package and are stored in the database directory as shown in Figure 18.4(*c*).

The directory structure shown in Figure 18.4(*c*) is the result of applying the architectural-style functional subsystem organization. It assumes that the software system supports multiple database management systems (DBMS). Therefore, the proxy classes and the database-access command classes are grouped into DBMS-specific subsystems such as Oracle, SQL Server, and LDAP subsystems. These are subsystems of the database subsystem. Accordingly, the classes in these subsystems have DBMS specific package names such as "database.oracle," "database.sqlserver," and "database.ldap." They are stored in the corresponding subdirectories under the database directory. The business objects form business-specific subsystems that often resemble their counterparts in the real world. For example, the business objects of a library information system can form circulation, cataloguing, purchasing, and user service subsystems. Therefore, classes belonging to these subsystems are stored in the circulation, cataloguing, purchasing, and usersvc subdirectories of the business objects directory. Similarly, the graphical user interface classes are stored in their respective subsystem's directories as shown in Figure 18.4(*c*). Organizing the classes into different subsystems, packages, and directories facilitates software reuse. For example, to reuse the database subsystem or one of its DBMS-specific components, one only needs to include the corresponding directory.

18.3 GENERATING CODE FROM DESIGN

As discussed in the introduction chapter, the object-oriented paradigm features seamless transition from analysis to design and from design to programming. Therefore, transition from design to implementation is relatively easy. Moreover, many integrated development environments (IDEs) can generate code skeletons from UML diagrams. However, depending on the implementation of the IDE, the code skeletons generated are different. If you don't use an IDE or don't want to use the code generated by an IDE, you can program yourself. This section presents rules for generating code skeletons from UML diagrams.

18.3.1 Implementing Classes and Interfaces

Mapping the classes and interfaces in the DCD to Java, C++, or C# is an easy and straightforward exercise because the modeling concepts correspond to the programming concepts nicely. Moreover, many IDEs can generate the classes and interfaces from UML class diagrams. It is therefore, not elaborated further in this book.

18.3.2 From Sequence Diagram to Method Code Skeleton

Sequence diagrams model the behavioral aspect of objects. As described in Chapter 11 (Deriving Design Class Diagram), the messages that are passed between the objects are used to identify operations of classes and relationships between the classes. This section illustrates the generation of the code skeletons for the functions of a class from a sequence diagram. Again, many IDEs provide this capability, but it is worthwhile to know the correspondence between design and implementation. Consider the sequence diagram in Figure 9.4. The long, narrow rectangle under the CheckoutController object indicates the execution of the checkout(callNo: String):String method of the CheckoutController. The arrow lines that go out from this rectangle represent calls to functions of other objects. The large box with “alt” at the upper-left corner implies a selection or if-then-else statement. Thus, from the sequence diagram, one can generate the following code skeleton for the CheckoutController class:

```
public class CheckoutController {
   Patron p;
   public String checkout(String callNo) {
      DBMgr dbm=new DBMgr();
      Document d=dbm.getDocument(callNo);
      String msg="";
      if (d != null) {
         Loan l=new Loan(p, d);
         dbm.save(l);
         d.setAvailable(false);
         dbm.save(d);
         msg="Checkout successful.";
      } else {
         msg="Document not found.";
      }
      return msg;
   }
}
```

Clearly, the above code skeleton is incomplete. The programmer needs to fill in the missing parts such as how to initialize the Patron object p and what packages need to be imported. But the ability to generate code skeleton from the sequence diagram greatly simplifies the programming task.

18.3.3 Implementing Association Relationships

Given a class diagram, the mapping of its classes, attributes, operations, inheritance, and implementing relationships to code is straightforward. An aggregation relationship, such as class B is an aggregate of class A, is implemented by a reference from B to A to represent the fact that an instance of A is part of an instance of B. There are several approaches to implement association relationships. One approach uses reference or pointer, described as follows:

One-to-one association. A one-to-one association between class A and class B is implemented by A holding a reference to B if A calls a function of B, and/or by B holding a reference to A if B calls a function of A.

One-to-many association. A one-to-many association between class A and class B is implemented by A holding a collection of references to B if A calls the functions of B instances, or by B holding a reference to A if instances of B call a function of A.

Many-to-many association. A many-to-many association between class A and class B is similarly implemented by a collection of references from A to B, and vice versa.

In the above, using a reference from A to B as well as a reference from B to A improves performance if A calls B and B calls A. However, adding or removing an instance of the association requires change to both references. Another approach to implement association relationships introduces a new class to represent the relationship. For example, an association between class A and class B is implemented by defining a class, say AssocAB, that has references to both A and B and uses a private collection to hold its instances:

```
public class AssocAB {
   private static Collection instances;
   private A a; private B b;
   public AssocAB(A a, B b) {
      this.a=a; this.b=b;
   }
   public void add(AssocAB ab) { ... }
   public void add(A a, B b) { ... }
   public void remove(AssocAB ab) { ... }
   ...
}
```

The private collection should be carefully chosen according to the needs of accessing the elements. For example, if the access is sequential, then using an array list is sufficient. If random access is needed, then using a hash table is more convenient and more efficient.

18.4 ASSIGNING IMPLEMENTATION WORK TO TEAM MEMBERS

During the implementation phase, the classes in the DCD are assigned to team members to implement and test. Usually, the classes depend on each other. Therefore, they should be implemented and tested in some order. For example, if class B calls a method of class A, then A should be implemented and tested before B. If B is implemented before A, then testing B requires the construction of a dummy class to simulate A. The class is called a test stub. The use of test stubs has several drawbacks. It consumes time and effort. If the behavior of A is complex, the construction of the test stub could be difficult, if not impossible. The test stub is not the class it simulates. Therefore, testing with stubs has limitations. Retesting is always required when A is implemented. Therefore, the classes of a DCD should be implemented and tested according to their dependencies. In general, classes that other classes depend on should be assigned to team members who can complete the implementation fast. This allows the team to proceed with the implementation work without needing to wait for the completion of the other classes.

18.5 PAIR PROGRAMMING

Pair programming is an emerging programming technique that requires two people to program together at one machine, with one keyboard and one mouse. The two programmers play two different roles but both work on the same program simultaneously. While the one with the keyboard and the mouse focuses on the best way to implement the functionality, the other reviews the program as it is being typed. For convenience, these two roles are referred to as the writer and the reviewer. They are also referred to as the driver and observer, or other similar roles in the literature. The partners switch roles periodically or whenever they like, for example, switching roles between programming sessions. Each programming session lasts about one to three hours. The pairs are not fixed, they switch around all the time. If a team adopts pair programming, then all team members must learn to work with others. If two people cannot work together, they don't have to pair with each other.

The team divides the programming work into small tasks and assigns them to the pairs. Each task can be a class, a use case, or a package. Using the methodology described in this book, assigning classes to pairs works well. The technique presented in Section 18.4 can be used to assign the classes to the pairs. It is important to maintain constant communication between the pairs to exchange progress, issues, and changes that are needed. With test-driven development, to be presented in the next section, the partners work together to write test cases, implement the functionality, run the tests, modify the tests or program, and refactor the code. The partners discuss what to do next, why they do that, and how they do it. The writer then implements the ideas and the reviewer checks the code for correctness, completeness, consistency, and provides improvement suggestions. Pair programming brings a number of benefits, as follows:

1. It reduces pressure and brings fun to programming because there is always someone with whom to share the stress and joy of success.
2. It enhances team communication because the partners exchange ideas during pair programming. Moreover, pair-switching helps the team members gain insight into the components of the system. This improves productivity and quality.
3. It enhances mutual understanding and collaboration because pair programming lets the team members understand each other and learn from each other in various ways and aspects.

4. It tends to produce simpler and more efficient solutions faster because discussions stimulate creative thinking and help in problem solving.

Pair programming has a number of limitations:

1. It is not for everyone—there are programmers who prefer to work alone.
2. If it is not handled properly, the discussion between the partners can take a lot of time. Therefore, the discussions should focus on solving the problem and getting the work done.
3. It could be slow to start due to the need to adapt to the differences of the partners in education, experience, problem-solving approach, and coding style.
4. Other limitations are: (1) the partners have to be at the same location, (2) it may be difficult for the partners to find a meeting time due to conflicting schedules, and (3) it might not be as effective as systematic review methods in detecting defects.

A programming practice that is similar to pair programming is ping-pong programming. With ping-pong programming, one developer writes the tests and the other developer implements the functionality. Ping-pong programming goes along very well with test-driven development, presented in the next section. Ping-pong programming is an iterative process. First, the basic tests and functionality are implemented. The boundary or extreme cases are added incrementally until the functionality and tests are complete.

18.6 TEST-DRIVEN DEVELOPMENT

Conventional approach to implementation writes the program before writing the tests. In recent years, test-driven development (TDD) is increasing its popularity in the software industry. TDD requires the programmer to "write tests first, then implement the functionality." To write the tests, the programmer must understand the functionality. Tests are generally written for the functions of a class. Recent studies indicate that TDD can reduce prerelease defect densities by as much as 90%, compared to other similar projects that do not implement TDD [116]. In addition, TDD improves programmer productivity [60]. TDD works well for pair programming as well as solo programming.

18.6.1 Test-Driven Development Workflow

Figure 18.5 shows the TDD workflow. With TDD, the features of a class are implemented and tested incrementally. To save time and effort, it is not necessary to test the ordinary get and set functions. TDD performs the following steps:

1. *Prepare for test-driven development.* In this step, the skeleton code for the class is generated. The test coverage to be achieved is determined. A test coverage is a quality criterion. For example, a 100% statement coverage means that the tests must execute each statement of the program at least once.
2. *Write tests.* In this step, the programmer selects the features to be implemented next. He also designs and implements the tests for the features.

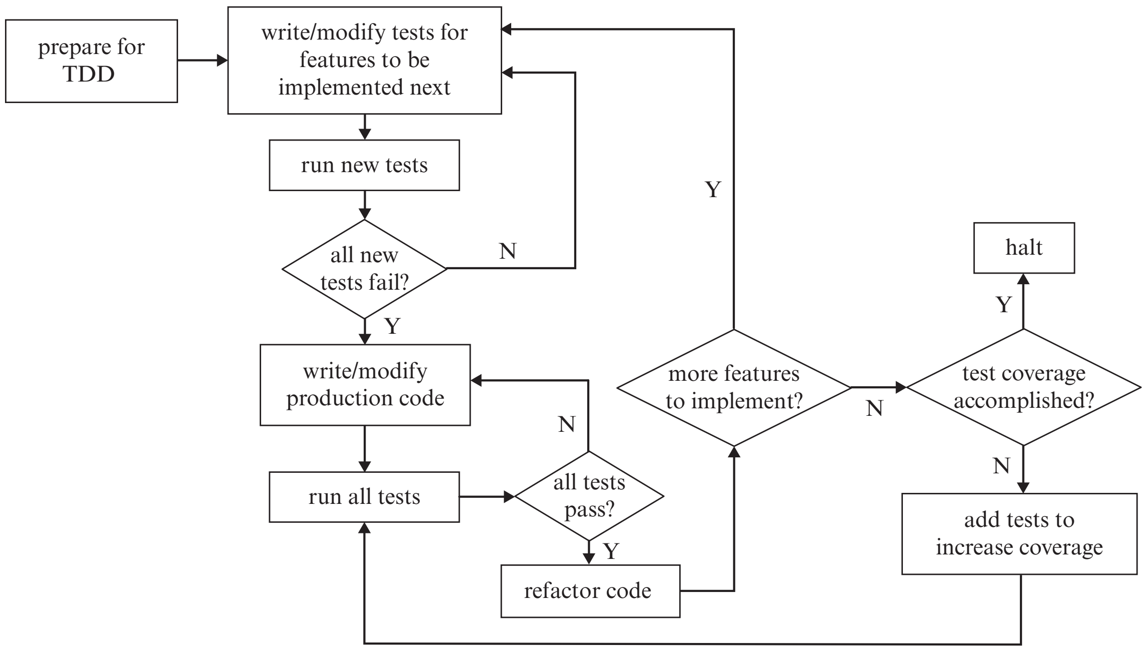

prepare for TDD
write/modify tests for features to be implemented next
run new tests
all new tests fail?
N
Y
write/modify production code
run all tests
all tests pass?
N
Y
refactor code
more features to implement?
Y
N
test coverage accomplished?
Y
halt
N
add tests to increase coverage

FIGURE 18.5 Test-driven development workflow

3. *Implement and test the features.* In this step, the programmer implements the features and runs the tests to ensure that the features are correctly implemented.
4. *Repeat until all features are correctly implemented.* Steps 2 and 3 are repeated until all the features are implemented and pass all the tests.
5. *Accomplish test coverage.* In this step, the programmer checks the test coverage, adds more tests, and modifies existing tests to ensure that the test-coverage objective is accomplished. The programmer may need to modify the program to pass the modified tests.

Step 1. Prepare for test-driven development.

Before TDD, the skeleton code for the class to be implemented is produced. It has all the functions including the get and set functions. Each function has an empty body, or returns a default value (null for returning an object). The class skeleton can be generated from the design class diagram (DCD). As a good software engineering practice, the skeleton code should include a file header that complies to adopted coding standards. In addition, the test coverage criteria are determined. Branch coverage is commonly used. It means that all branches of the class must be tested at least once. For example, if a program contains only one if-then-else statement, then there are two branches because the condition has two outcomes. To achieve branch coverage, at least two tests are required. They must ensure that the program executes each branch correctly.

Step 2. Select features and write tests for the features.

In this step, the programmer selects the features to be implemented next. Often, features that realize high-priority requirements should be implemented first.

A requirements traceability tool is useful in this regard. For example, during the requirements phase, requirements are ranked according to the customer's business priorities. The requirement priorities determine the priorities of the use cases since use cases are derived from requirements. The use case priorities, in turn, determine the priorities of the functions that are assigned to the objects during object interaction modeling. Thus, the functions of a class in the design class diagram are prioritized. The priorities of the functions of a class may differ because they are derived from different use cases. A traceability tool should automatically track and capture such information during the development process. The selection of the features must also consider the dependencies among the features. For example, if function f2 calls function f1, or f2 uses data produced by f1, then f1 should be implemented no later than f2.

Once the features are selected, the programmer designs the tests according to the functionality of the features. Using a unit test tool such as JUnit, the programmer implements the tests. Appendix C contains a brief tutorial on how to use JUnit. The tests are run to test the features. Since the functionality is not implemented, the program should fail to pass each of the tests. If it is not the case, then the tests need to be strengthened and rerun until the program fails to pass the tests.

Step 3. Implement and test the features.

In this step, the programmer implements the features according to their functionality and runs the tests. The programmer may need to modify the program as well as the tests to remove errors. The programmer performs these activities until all the selected features are correctly implemented. The programmer then cleans up the production code as well as the test code. He or she also improves the structure, readability, and performance of the program. In-code documentation and comments are added as required by the codeing standards. These activities are called *refactoring*.

Step 4. Iterate until all features are implemented.

Steps 2 to 3 are repeated until all features of the class are implemented and the program passes all the tests.

Step 5. Ensure test coverage.

The programmer measures the test coverage using a coverage tool such as Emma or Cobertura, described in Appendix C. The programmer adds tests and reruns all the tests until the required test coverage is accomplished. During this step, refactoring is also performed.

18.6.2 Merits of Test-Driven Development

TDD is gaining popularity in the software industry because it brings a number of benefits to software development:

1. TDD requires the programmer to understand the functionality and implement testable features. Testability is an important attribute of software, especially software requirements. In this sense, TDD helps the team understand and improve the requirements.

2. TDD constantly validates the implementation with respect to the tests. It helps the team detect and remove defects. As a result, TDD produces high-quality code.
3. TDD focuses on the desired functionality first but also addresses the other quality aspects such as program structure and readability through refactoring.
4. TDD facilitates debugging because incremental implementation of the features makes it easy to locate and fix errors.

18.6.3 Potential Problems

There are a number of potential problems, which should be noted, although they are not specific to TDD:

1. The test cases may be too weak to ensure that the program indeed correctly implements the desired functionality. For example, the test cases only test that an object is stored in the database; it does not check that the attribute values are correctly stored.
2. The test cases may be too focused on the main functionality and overlook other cases that may cause the program to crash or behave incorrectly. For example, storing a null object into the database or an object with illegal attribute values should be tested but could be missing from the test cases.
3. The test cases or test scripts are themselves programs. If they are not written in accordance to coding standards and conventions, then the maintenance of these programs is a nightmare. Therefore, the refactoring activity should include refactoring of these programs. Moreover, the subject class and the test cases should comply to coding standards.

18.7 APPLYING AGILE PRINCIPLES

GUIDELINE 18.8 Develop small, incremental releases and iterate; focus on frequent delivery of software products.

Agility favors the development of small, incremental releases iteratively. This greatly reduces the risk of requirements misconception, a significant factor of project failure. Small, incremental releases make it easy to change the software to respond to users' feedback. The practice reduces the users' learning curve. As the increments are successfully deployed, the positive feeling of the team and the users increases. This, in turn, strengthens the collaboration and cooperation of the team and users.

GUIDELINE 18.9 Complete each feature before moving onto the next.

It is important to complete each feature before moving onto the next because agile development values working software. Implementing this principle means only completed features are counted, not partially completed features because the latter are not working software.

GUIDELINE 18.10 Test early and often.

Testing early and often allows the team to detect errors early and correct them much more easily. Test early and often applies to unit testing, as well as integration testing and acceptance testing. Modern tools such as IDEs and JUnit make it easy to implement this principle. For example, hundreds of tests can be run easily with such tools in just a few seconds. Frequent testings greatly improve the quality of the software.

GUIDELINE 18.11 Everybody owns the code—everybody is responsible for the quality of the code.

Collective code ownership improves teamwork because it encourages the team members to contribute to all program segments of the project. For example, extreme programming advocates anybody can change any line of code at any time. Collective code ownership requires that all code that is released into the source code repository must include unit tests that run at 100%. This implies that anyone who changes the code must ensure that the modified code also includes unit tests that run at 100%. Collective code ownership implements the agile principle that empowers the team to make decisions. Through this practice, the team members share their knowledge of the components of the system. It improves the team member's understanding of the components. It also reduces the risk of a team member leaving the project.

18.8 TOOL SUPPORT FOR IMPLEMENTATION

NetBeans and Eclipse are widely used tools in the implementation phase. They are available from http://netbeans.org/ and www.eclipse.org/, respectively. These are two of the many integrated development environments (IDEs) that support many of the software development activities. These tools allow the programmer to create, edit, and manage program files and tests for multiple projects, compile and execute the programs and tests, track the test coverages, and debug the programs and tests as well as version control. Appendix C describes these tools in more detail.

SUMMARY

This chapter presents implementation considerations. These include coding standards, organization of implementation artifacts, converting design to programs, assigning implementation work to team members, and how to conduct test-driven development, pair programming, and ping-pong programming. The chapter also discusses applying agile principles during the implementation phase and tool support to implementation and test-driven development.

FURTHER READING

For more information on pair programming, the reader is referred to [162]. Pair programming also works for extreme programming, an emerging software development paradigm. There are many good books on extreme programming, and test-driven development, see for example, [14, 19, 20, 21, 97]. The agile alliance website at www.agilealliance.org contains many articles on these agile development topics. Many good books on various object-oriented programming languages exist in the market. These include [13, 83, 84, 138, 141, 142, 152]. Appendix B provides an introduction to Java, Java database connectivity (JDBC), Swing, and Java server pages (JSPs). More detail about these Java technologies are available from Oracle's website, which is also a good source for other useful information such as Java coding standards and software download. The implementation and test order for the classes in the DCD was first proposed in [99, 101]. Other algorithms on computing an optimal order are found in [37, 38, 103, 147, 151]. However, the problem of finding an optimal order remains an open research problem although a good enough algorithm is practically good enough.

CHAPTER REVIEW QUESTIONS

1. What are coding standards, and why are they important?
2. How are the design diagrams mapped to code?
3. What is test-driven development, its workflow, benefits, and potential problems?
4. What is pair programming, its benefits and limitations?
5. How does one assign the classes of a design class diagram to the team members to implement?
6. How does one apply agile principles during implementation?

EXERCISES

Chapter 16 describes the design of a state diagram editor. It produces a number of design diagrams. The following three exercises are related to the design of the editor. Each program file must include the file header and description of classes using the format in Figures 18.2 and 18.3.

18.1 Produce a directory structure for the state diagram editor and assign the editor-related classes presented in Chapter 16 to the directories and subdirectories.

18.2 Implement the classes in Figure 16.6 using test-driven development. Use JUnit to write and run the test cases, and Cobertura to ensure that 100% branch coverage is accomplished.

18.3 Implement the classes in Figure 16.26 using test-driven development and pair programming. Form the pair yourselves or according to the instructions given by the instructor. Use JUnit to write and run the test cases, and Cobertura to ensure that 100% branch coverage is accomplished.

18.4 Implement the persistence framework presented in Chapter 17. Assume that the application is a library information system and the database is a local database. Do this exercise using test-driven development, pair programming, and JDBC, or as instructed by the instructor. In addition, assume that the classes in Figure 11.10 have the following attributes:

Document(cn: String, title: String, author: String, publisher: String, ISBN: String, year: String, duration: String)

Loan(patron: Patron, document: Document, date: String, dueDate: String), the date has the format as "MM/DD/YYYY."

Patron(id: String, name: String, address: String, tel: String)

18.5 Implement the design of the business rules class diagram you produced in exercise 15.7.

18.6 Implement the design of the lawn mowing agent you produced in exercise 13.6.

18.7 Implement the graphical user interface for the state diagram editor presented in Chapter 16.

18.8 Design and implement a web page or two that allows the user to search for documents in a library information system. Do this exercise based on the persistence framework you produced in the above exercise. Use JSP or a technology as instructed by your instructor. An introduction to JSP is given in Appendix B. *Hint:* You need to design and implement a search use case controller that interacts with the DBMgr. You also need to extend the persistence framework to provide the search capability.