Gotos: Gotos should be used sparingly and in a disciplined manner. Only when the alternative to using gotos is more complex should the gotos be used. In any case, alternatives must be thought of before finally using a goto. If a goto must be used, forward transfers (or a jump to a later statement) is more acceptable than a backward jump.

Information Hiding: As discussed earlier, information hiding should be supported where possible. Only the access functions for the data structures should be made visible while hiding the data structure behind these functions.

User-Defined Types: Modern languages allow users to define types like the enumerated type. When such facilities are available, they should be exploited where applicable. For example, when working with dates, a type can be defined for the day of the week. Using such a type makes the program much clearer than defining codes for each day and then working with codes.

Nesting: If nesting of if-then-else constructs becomes too deep, then the logic become harder to understand. In case of deeply nested if-then-elses, it is often difficult to determine the if statement to which a particular else clause is associated. Where possible, deep nesting should be avoided, even if it means a little inefficiency. For example, consider the following construct of nested if-then-elses:

```
if C1 then S1
    else if C2 then S2
        else if C3 then S3
            else if C4 then S4;
```

If the different conditions are disjoint (as they often are), this structure can be converted into the following structure:

```
if C1 then S1;
if C2 then S2;
if C3 then S3;
if C4 then S4;
```

This sequence of statements will produce the same result as the earlier sequence (if the conditions are disjoint), but it is much easier to understand. The price is a little inefficiency.

Module Size: We discussed this issue during system design. A programmer should carefully examine any function with too many statements (say more than 100). Large modules often will not be functionally cohesive. There can be no hard-and-fast rule about module sizes the guiding principle should be cohesion and coupling.

Module Interface: A module with a complex interface should be carefully examined. As a rule of thumb, any module whose interface has more than five parameters should be carefully examined and broken into multiple modules with a simpler interface if possible.

Side Effects: When a module is invoked, it sometimes has side effects of modifying the program state beyond the modification of parameters listed in the module interface definition, for example, modifying global variables. Such side effects should be avoided where possible, and if a module has side effects, they should be properly documented.

Robustness: A program is robust if it does something planned even for exceptional conditions. A program might encounter exceptional conditions in such forms as incorrect input, the incorrect value of some variable, and overflow. If such situations do arise, the program should not just "crash" or "core dump"; it should produce some meaningful message and exit gracefully.

Switch case with default: If there is no default case in a "switch" statement, the behavior can be unpredictable if that case arises at some point of time which was not predictable at development stage. Such a practice can result in a bug like NULL dereference, memory leak, as well as other types of serious bugs. It is a good practice to always include a default case.

```
switch (i){
      case 0 : {s=malloc(size)
      }
 s[0] = y; /* NULL dereference if default occurs*/
```

Empty Catch Block: An exception is caught, but if there is no action, it may represent a scenario where some of the operations to be done are not performed. Whenever exceptions are caught, it is a good practice to take some default action, even if it is just printing an error message.

```
try {
      FileInputStream fis = new
      FileInputStream("InputFile");
}
catch (IOException ioe) { }
      // not a good practice
```

Empty if, while Statement: A condition is checked but nothing is done based on the check. This often occurs due to some mistake and should be caught. Other similar errors include empty finally, try, synchronized, empty static method, etc. Such useless checks should be avoided.

```
if (x == 0) {}  /* nothing is done after checking x */
else {
   :
}
```

Read Return to be Checked: Often the return value from reads is not checked, assuming that the read returns the desired values. Sometimes the result from a read can be different from what is expected, and this can cause failures later. There may be some cases where neglecting this condition may result in some serious error. For example, if read from scanf() is more than expected, then it may cause a buffer overflow. Hence the value of read should be checked before accessing the data read. (This is the reason why most languages provide a return value for the read operation.)

Return From Finally Block: One should not return from finally block, as cases it can create false beliefs. For example, consider the code

```
public String foo() {
      try {
             throw new Exception( "An Exception" );
      }
      catch (Exception e) {
             throw e;
      }
finally {
             return "Some value";
             }
      }
```

In this example, a value is returned both in exception and nonexception scenarios. Hence at the caller site, the user will not be able to distinguish between the two. Another interesting case arises when we have a return from try block. In this case, if there is a return in finally also, then the value from finally is returned instead of the value from try.

Correlated Parameters: Often there is an implicit correlation between the parameters. For example, in the code segment given below, "length" represents the size of BUFFER. If the correlation does not hold, we can run into a serious problem like buffer overflow (illustrated in the code fragment below). Hence, it is a good practice to validate this correlation rather than assuming that it holds. In general, it is desirable to do some counter checks on implicit assumptions about parameters.

```
  void (char *src, int length, char destn[]) {
strcpy (destn, src); /* Can cause buffer overflow
                        if length > MAX_SIZE */
  }
```

Trusted Data sources: Counter checks should be made before accessing the input data, particularly if the input data is being provided by the user or is being obtained over the network. For example, while doing the string copy operation, we should check that the source string is null terminated, or that its size is as we expect. Similar is the case with some network data which may be sniffed and prone to some modifications or corruptions. To avoid problems due to these changes, we should put some checks, like parity checks, hashes, etc. to ensure the validity of the incoming data.

Give Importance to Exceptions: Most programmers tend to give less attention to the possible exceptional cases and tend to work with the main flow of events, control, and data. Though the main work is done in the main path, it is the exceptional paths that often cause software systems to fail. To make a software system more reliable, a programmer should consider all possibilities and write suitable exception handlers to prevent failures or loss when such situations occur.

9.1.5 Coding Standards

Programmers spend far more time reading code than writing code. Over the life of the code, the author spends a considerable time reading it during debugging and enhancement. People other than the author also spend considerable effort in reading code because the code is often maintained by someone other than the author. In short, it is of prime importance to write code in a manner that it is easy to read and understand. Coding standards provide rules and guidelines for some aspects of programming in order to make code easier to read. Most organizations who develop software regularly develop their own standards.

In general, coding standards provide guidelines for programmers regarding naming, file organization, statements and declarations, and layout and comments. To give an idea of coding standards (often called conventions or style guidelines), we discuss some guidelines for Java, based on publicly available standards (from www.geosoft.no or java.sun.com/docs).

Naming Conventions

Some of the standard naming conventions that are followed often are:

- Package names should be in lower case (e.g., mypackage, edu.iitk.maths)
- Type names should be nouns and should start with uppercase (e.g., Day, DateOfBirth, EventHandler)
- Variable names should be nouns starting with lower case (e.g., name, amount)
- Constant names should be all uppercase (e.g., PI, MAX_ITERATIONS)
- Method names should be verbs starting with lowercase (e.g., getValue())

- Private class variables should have the _ suffix (e.g., "private int value_"). (Some standards will require this to be a prefix.)

- Variables with a large scope should have long names; variables with a small scope can have short names; loop iterators should be named i, j, k, etc.

- The prefix *is* should be used for boolean variables and methods to avoid confusion (e.g., isStatus should be used instead of status); negative boolean variable names (e.g., isNotCorrect) should be avoided.

- The term *compute* can be used for methods where something is being computed; the term *find* can be used where something is being looked up (e.g., computeMean(), findMin().)

- Exception classes should be suffixed with *Exception* (e.g., OutOfBoundException.)

Files

There are conventions on how files should be named, and what files should contain, such that a reader can get some idea about what the file contains. Some examples of these conventions are:

- Java source files should have the extension .java—this is enforced by most compilers and tools.

- Each file should contain one outer class and the class name should be same as the file name.

- Line length should be limited to less than 80 columns and special characters should be avoided. If the line is longer, it should be continued and the continuation should be made very clear.

Statements

These guidelines are for the declaration and executable statements in the source code. Some examples are given below. Note, however, that not everyone will agree to these. That is why organizations generally develop their own guidelines that can be followed without restricting the flexibility of programmers for the type of work the organization does.

- Variables should be initialized where declared, and they should be declared in the smallest possible scope.

- Declare related variables together in a common statement. Unrelated variables should not be declared in the same statement.
- Class variables should never be declared public.
- Use only loop control statements in a for loop.
- Loop variables should be initialized immediately before the loop.
- Avoid the use of *break* and *continue* in a loop.
- Avoid the use of *do ... while* construct.
- Avoid complex conditional expressions – introduce temporary boolean variables instead.
- Avoid executable statements in conditionals.

Commenting and Layout

Comments are textual statements that are meant for the program reader to aid the understanding of code. The purpose of comments is not to explain in English the logic of the program—if the logic is so complex that it requires comments to explain it, it is better to rewrite and simplify the code instead. In general, comments should explain what the code is doing or why the code is there, so that the code can become almost standalone for understanding the system. Comments should generally be provided for blocks of code, and in many cases, only comments for the modules need to be provided.

Providing comments for modules is most useful, as modules form the unit of testing, compiling, verification and modification. Comments for a module are often called *prologue* for the module, which describes the functionality and the purpose of the module, its public interface and how the module is to be used, parameters of the interface, assumptions it makes about the parameters, and any side effects it has. Other features may also be included. It should be noted that prologues are useful only if they are kept consistent with the logic of the module. If the module is modified, then the prologue should also be modified, if necessary.

Java provides *documentation comments* that are delimited by "/** ... */", and which could be extracted to HTML files. These comments are mostly used as prologues for classes and its methods and fields, and are meant to provide documentation to users of the classes who may not have access to the source code. In addition to prologue for modules, coding standards may specify how and where comments should be located. Some such guidelines are:

- Single line comments for a block of code should be aligned with the code they are meant for.

- There should be comments for all major variables explaining what they represent.
- A block of comments should be preceded by a blank comment line with just "/*" and ended with a line containing just "*/".
- Trailing comments after statements should be short, on the same line, and shifted far enough to separate them from statements.

Layout guidelines focus on how a program should be indented, how it should use blank lines, white spaces, etc. to make it more easily readable. Indentation guidelines are sometimes provided for each type of programming construct. However, most programmers learn these by seeing the code of others and the code fragments in books and documents, and many of these have become fairly standard over the years. We will not discuss them further except saying that a programmer should use some conventions, and use them consistently.

9.2 Coding Process

The coding activity starts when some form of design has been done and the specifications of the modules to be developed are available. With the design, modules are usually assigned to individual developers for coding. In a top-down implementation, we start by assigning modules at the top of the hierarchy and proceed to the lower levels. In a bottom-up implementation, the development starts with first implementing the modules at the bottom of the hierarchy and proceeds up. The impact of how we proceed is on integration and testing.

When modules are assigned to developers, they use some process for developing the code. We now look at some processes that developers use during coding, or that have been suggested.

9.2.1 An Incremental Coding Process

The process followed by many developers is to write the code for the currently assigned module, and when done, perform unit testing on it and fix the bugs found. Then the code is checked in the project repository to make it available to others in the project. (We will explain the process of checking in later.)

A better process for coding, that is often followed by experienced developers, is to develop the code incrementally. That is, write code for implementing only part of the functionality of the module. This code is compiled and tested with some quick tests to check the code that has been written so far. When the code passes these tests, the developer proceeds to add further functionality to the code, which is then tested again. In other words, the code is built incrementally by the developers, testing it as it is built. This coding process is shown in Figure 9.2.

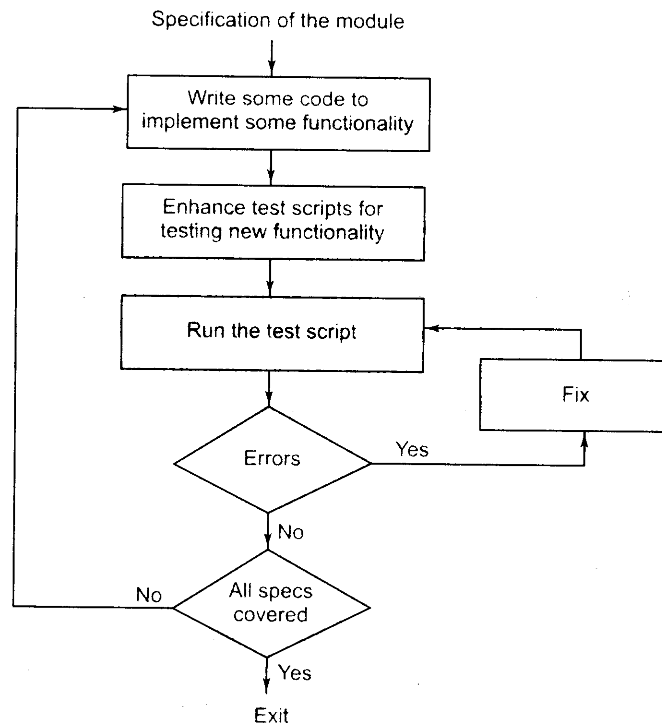

Figure 9.2: An incremental coding process.

The basic advantage of developing code incrementally with testing being done after every round of coding is to facilitate debugging—an error found in some testing can be safely attributed to code that was added since last successful testing. For following this process, it is essential that there be automated test scripts that can run the test cases with the click of a button. With these test scripts, testing can be done as frequently as desired, and new test cases can be added easily. These test scripts are a tremendous aid when code is enhanced in future due to requirement changes—through the test scripts it can be quickly checked that the earlier functionality is still working. These test scripts can also be used with some enhancements for the final unit testing that is often done before checking in the module.

9.2.2 Test Driven Development

Test Driven Development (TDD) [11] is a coding process that turns around the common approach to coding. In TDD, a programmer first writes the test scripts, and then writes the code to pass the tests. The whole process is done incrementally, with tests being

written based on the specifications and code being written to pass the tests. The TDD process is shown in Figure 9.3.

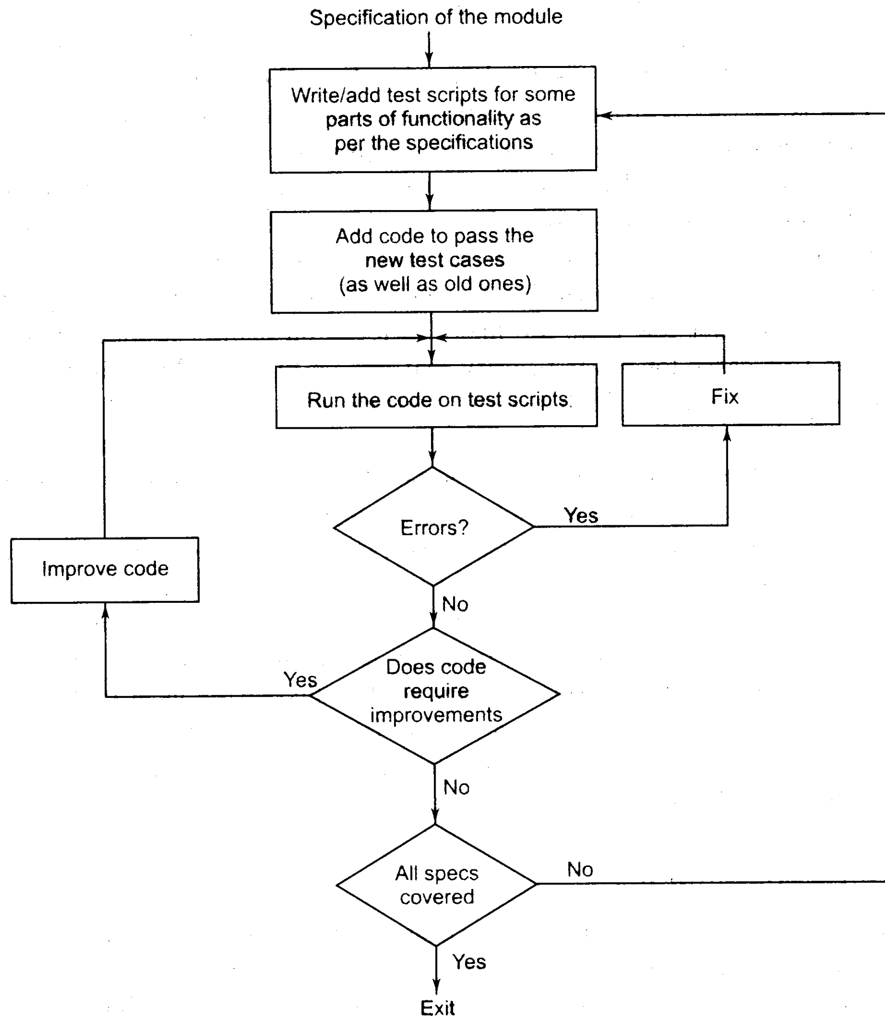

Figure 9.3: Test driven development process.

This is a relatively new approach, which has been adopted in the extreme programming (XP) methodology [10]. The concept of TDD is, however, general and not tied to any particular methodology. The discussion of TDD here is based on [11].

A few points are worth noting about TDD. First, the approach says that you write just enough code to pass the tests. By following this, the code is always in sync with the tests. This is not always the case with the code-first approach, in which it is all

too common to write a long piece of code, but then only write a few tests which cover only some parts of the code. By encouraging that code is written only to pass the tests, the responsibility of ensuring that required functionality is built is being passed to the activity of writing the test cases. That is, it is the task of test cases to check that the code that will be developed has all the functionality needed.

This writing of test cases before the code is written makes the development usage-driven. That is, first the focus is to determine how the code to be developed will be used. This is extracted from the specifications and the usage interface is specified precisely when the test cases are written. This helps ensure that the interfaces are from the perspective of the user of the code and that some key usage scenarios have been enunciated before the code is written. The focus is on the users of the code and the code is written to satisfy the users. This can reduce interface errors.

In TDD, some type of prioritization for code development naturally happens. It is most likely that the first few tests are likely to focus on using the main functionality. Generally, the test cases for lower priority features or functionality will be developed later. Consequently, code for high priority features will be developed first and lower priority items will be developed later. This has the benefit that higher priority items get done first, but has the drawback that some of the lower priority features or some special cases for which test cases are not written may not get handled in the code.

As the code is written to satisfy the test cases, the completeness of the code depends on the thoroughness of the test cases. Often it is hard and tedious to write test cases for all the scenarios or special conditions, and it is highly unlikely that a developer will write test cases for all the special cases. In TDD, as the goal is to write enough code to pass the test cases, such special cases may not get handled. Also, as at each step code is being written primarily to pass the tests, it may later be found that earlier algorithms were not well suited. In that case, the code should be improved before new functionality is added, as shown in Figure 9.3.

9.2.3 Pair Programming

Pair programming is also a coding process that has been proposed as a key technique in extreme programming (XP) methodology [10]. In pair programming, code is not written by individual programmers but by a pair of programmers. That is, the coding work is assigned not to an individual but to a pair of individuals. This pair together writes the code.

The process envisaged is that one person will type the program while the other will actively participate and constantly review what is being typed. When errors are noticed, they are pointed out and corrected. When needed, the pair discuss the algorithms, data structures, or strategies to be used in the code to be written. The roles are rotated frequently making both equal partners and having similar roles.

The basic motivation for pair programming is that as code reading and code reviews have been found to be very effective in detecting defects, by having a pair do the

programming we have the situation where the code is getting reviewed as it is being typed. That is, instead of writing code and then getting it reviewed by another programmer, we have a programmer who is constantly reviewing the code being written. Like incremental development and testing, we now have incremental reviewing taking place.

Besides ongoing code review, having two programmers apply themselves to the programming task at hand is likely to result in better decisions being taken about the data structures, algorithms, interfaces, logic, etc. Special conditions, which frequently result in errors, are also more likely to be dealt with in a better manner.

The potential drawback of pair programming is that it may result in loss of productivity by assigning two people for a programming task. It is clear that a pair will produce better code as compared to code being developed by a single programmer. The open question is whether this increase in productivity due to improved code quality offsets the loss incurred by putting two people on a task. There are also issues of accountability and code ownership, particularly when the pairs are not fixed and rotate (as has been proposed in XP). Impact of pair programming is an active area of research, particularly for experimental software engineering.

9.2.4 Source Code Control and Build

In a project many different people develop source code. Each programmer creates different source files, which are eventually combined together to create executables. Programmers keep changing their source files as the code evolves, as we have seen in the processes discussed above, and often make changes in other source files as well. In order to keep control over the sources and their evolution, source code control is almost always used in projects using tools like the CVS on UNIX (www.cvshome.org) or visual source safe (VSS) on Windows (msdn.microsoft.com/vstudio/previous/ssafe) Here we give a brief description of how these tools are used in the coding process. Earlier in Chapter 2 we have discussed the concepts of a general CM process. Our discussion is based on CVS.

A modern source code control system contains a repository, which is essentially a controlled directory structure, which keeps the full revision history of all the files. For efficiency, a file history is generally kept as deltas or increments from the base file. This allows any older version of the file to be recreated, thereby giving the flexibility to easily discard a change, should the need arise. The repository is also the "official" source for all the files.

For a project, a repository has to be set up with permissions for different people in the project. The files the repository will contain are also specified—these are the files whose evolution the repository maintains. Programmers use the repository to make their source files changes available, as well as obtain other source files. Some of the types of commands that are generally performed by a programmer are:

Get a local copy. A programmer in a project works on a local copy of the file. Com-

mands are provided to make a local copy from the repository. Making a local copy is generally called a *checkout*. An example command is *cvs checkout* $<$ *module* $>$, which copies a set of files that belongs to the $<$ *module* $>$ on the local machine. A user will get the latest copy of the file. However, if a user wants, any older version of a file can be obtained from the repository, as the complete history is maintained. Many users can check out a file.

Make changes to file(s). The changes made to the local file by a programmer remain local until the changes are *committed* back on the repository. By committing (e.g., by *cvs commit* $<$ *file* $>$) the changes made to the local file are made to the repository, and are hence available to others. This operation is also referred to as *check in*.

Update a local copy. Changes committed by project members to the repository are not reflected in the local copies that were made before the changes were committed. For getting the changes, the local copies of the files have to be updated (e.g., by *cvs update* command). By an update, all the changes made to the files are reflected in the local copy.

Get Reports. Source control tools provide a host of commands to provide different reports on the evolution of the files. These include reports like the difference between the local file and the latest version of the file, all changes made to a file along with the dates and reasons for change (which are typically provided while committing a change).

Note that once the changes are committed, they become available to all members of the team, who are generally supposed to use the source files from the repository for checking their own programs. Hence, it is essential that a programmer commits a source file only when it is in a state that it is usable by others. In steady state, the normal behavior of a project member will be as follows: check out the latest version of the files to be changed; make the planned changes to them; validate that the changes have the desired effect (for which all the files may be copied and the system tried out locally); commit the changes back to the repository.

It should be clear that if two people check out some file and then make changes, there is a possibility of a conflict—different changes are made to the same parts of the file. All tools will detect the conflict when the second person tries to commit the changes, and will inform the user. The user has to manually resolve the conflit, i.e., make the file such that the changes do not conflict with existing changes, and then commit the file. Conflicts are usually rare as they occur only if different changes are made to the same lines in a file.

With a source code control system, a programmer does not need to maintain all the versions—at any time if some changes need to be undone, older versions can be easily recovered. The repositories are always backed up, so they also provide protection

against accidental loss. Furthermore, a record of changes is maintained—who made the change and when, why was the change made, what were the actual changes, etc. Most importantly, the repository provides a central place for the latest and authoritative files of the project. This is invaluable for products that have a long life and that evolve over many years.

Besides using the repository for maintaining the different versions, it is also used for constructing the software system from the sources—an activity often called *build.* The build gets the latest version (or the desired version number) of the sources from the repository, and creates the executables from the sources.

Building the final executables from the source files is often done through tools like the Makefile [62], which specify the dependence between files and how the final executables are constructed from the source files. These tools are capable of recognizing that files have changed and will recompile whenever files are changed for creating the executables. With source code control, these tools will generally get the latest copy from the repository, then use it for creating executables.

This is one of the simplest approaches to source code control and build. Often, when large systems are being built, more elaborate methods for source code control and build are needed. Such methods often have a hierarchy of controlled areas, each having different levels of control and different sources, with the top of the hierarchy containing all the files needed to build the "official" system. Lower levels of the hierarchy can be used by different groups to create "local" builds for testing and other purposes. In such a system, forward integration and reverse integration is needed to pass changes back and forth between the controlled areas at different levels of the hierarchy. An advanced tool like ClearCase provides such capabilities.

9.3 Refactoring

We have seen that coding often involves making changes to some existing code. Code also changes when requirements change or when new functionality is added. Due to the changes being done to modules, even if we started with a good design, with time we often end up with code whose design is not as good as it could be. And once the design embodied in the code becomes complex, then enhancing the code to accommodate required changes becomes more complex, time consuming, and error prone. In other words, the productivity and quality starts decreasing.

Refactoring is the technique to improve existing code and prevent this design decay with time. Refactoring is part of coding in that it is performed during the coding activity, but is not regular coding. Refactoring has been practiced in the past by programmers, but recently it has taken a more concrete shape, and has been proposed as a key step in the Extreme Programming practice [10]. Refactoring also plays an important role in test driven development—code improvement step in the TDD process is really doing

refactoring. Here we discuss some key concepts and some methods for doing refactoring. The discussion here is based on the book on this topic by Fowler [65].

9.3.1 Basic Concepts

Refactoring is defined as a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [65]. A key point here is that the change is being made to the design embodied in the source code (i.e., the internal structure) exclusively for improvement purposes.

The basic objective of refactoring is to improve the design. However, note that this is not about improving a design during the design stages for creating a design which is to be later implemented (which is the focus of design methodologies), but about improving the design of code that already exists. In other words, refactoring, though done on source code, has the objective of improving the design that the code implements. Therefore, the basic principles of design guide the refactoring process. Consequently, a refactoring generally results in one or more of the following:

1. Reduced coupling
2. Increased cohesion
3. Better adherence to open-closed principle (for OO systems)

Refactoring involves changing the code to improve one of the design properties, while keeping the external behavior the same. Refactoring is often triggered by some coding changes that have to be done. If some enhancements are to be made to the existing code, and it is felt that if the code structure was different (better) then the change could have been done easier, that is the time to do refactoring to improve the code structure.

Even though refactoring is triggered by the need to change the software (and its external behavior), it should not be confused or mixed with the changes for enhancements. It is best to keep these two types of changes separate. So, while developing code, if refactoring is needed, the programmer should cease to write new functionality, and first do the refactoring, and then add new code.

The main risk of refactoring is that existing working code may "break" due to the changes being made. This is the main reason why most often refactoring is not done. (The other reason is that it may be viewed as an additional and unnecessary cost.) To mitigate this risk, the two golden rules are:

1. Refactor in small steps
2. Have test scripts available to test existing functionality

If a good test suite is available, then whether refactoring preserves existing functionality can be checked easily. Refactoring cannot be done effectively without an automated

test suite as without such a suite determining if the external behavior has changed or not will become a costly affair. By doing refactoring in a series of small steps, and testing after each step, mistakes in refactoring can be easily identified and rectified. With this, each refactoring makes only a small change, but a series of refactorings can significantly transform the program structure.

With refactoring, code becomes continuously improving. That is, the design, rather than decaying with time, evolves and improves with time. With refactoring, the quality of the design improves, making it easier to make changes to the code as well as find bugs. The extra cost of refactoring is paid for by the savings achieved later in reduced testing and debugging costs, higher quality, and reduced effort in making changes.

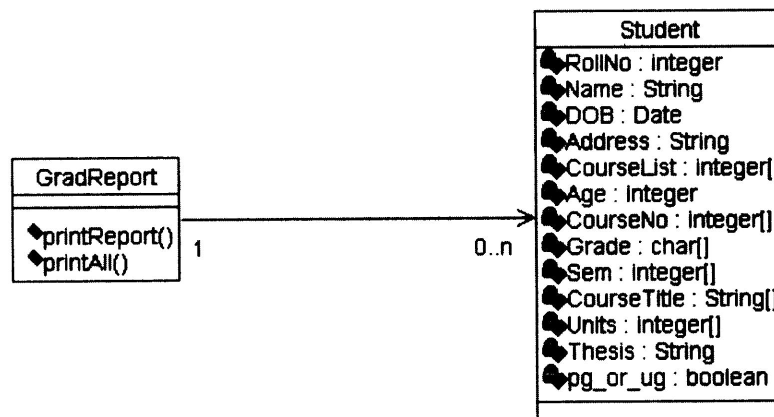

Figure 9.4: Initial class diagram.

If refactoring is to be practiced, its usage can also ease the design task in the design stages. Often the designers spend considerable effort in trying to make the design as good as possible, try to think of future changes and try to make the design flexible enough to accommodate all types of future changes they can envisage. This makes the design activity very complex, and often results in complex designs. With refactoring, the designer does not have to be terribly worried about making the best or most flexible design—the goal is to try to come up with a good and simple design. And later if new changes are required that were not thought of before, or if shortcomings are found in the design, the design is changed through refactoring. More often than not, the extra flexibility envisaged and designed is never needed, resulting in a system that is unduly complex.

Note that refactoring is not a technique for bug fixing or for improving code that is in very bad shape. It is done to code that is mostly working—the basic purpose is to

make the code live longer by making its structure healthier. It starts with healthy code and instead of letting it become weak, it continues to keep it healthy.

9.3.2 An example

Let us illustrate the refactoring process by an example. Let us consider a simplified system to produce a graduation report for a student. A student in a university takes a set of courses, and perhaps, writes a thesis. This system checks the whether a student has completed the graduation requirements, and prints the result along with the list of courses the students has taken, the thesis the student may have done, the student's cumulative grade points (referred to as CPI or cumulative point index), and other information about the student. A student may be a graduate student (referred to as PG or postgraduate) or an undergraduate (UG). To keep the example simple, the graduation requirements for the two are only in terms of number of courses they have to take, and that graduate students have to do a thesis.

Consider a simple implementation for this, whose design is shown in the class diagram in Figure 9.4. (The full code for this implementation as well as code after refactorings is available on the book's Web site.)

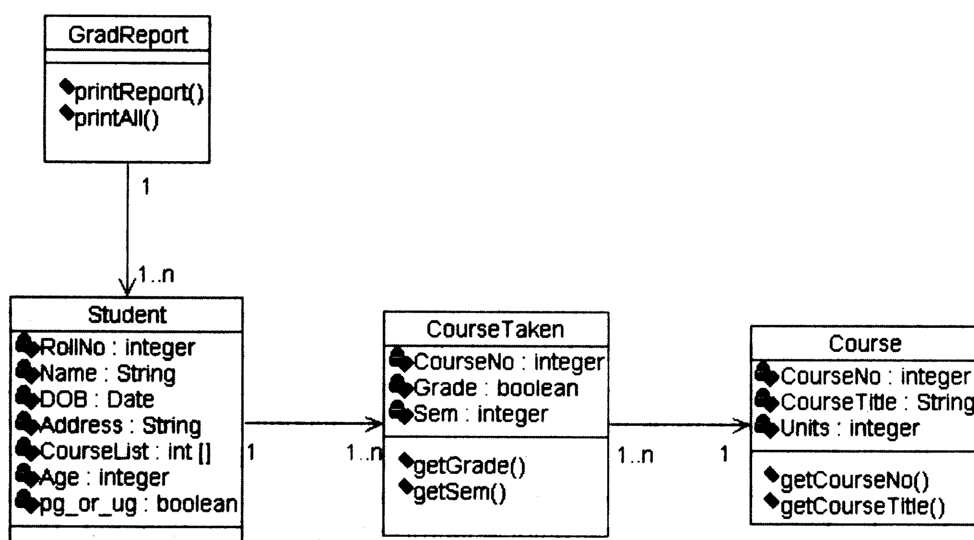

Figure 9.5: Class diagram after first refactoring.

There is an array of student objects, which is accessed by printReport(). The object for a students contains all the information about the student, which also provides a host

of methods (not shown in the diagram) to provide the required information to a client object. All the logic for producing the report is in PrintReport()—it gets info about the student, perform suitable checks depending on whether the student is a UG or a PG, and prints the data about the student, the report, the list of courses, and computes and prints the grade point (called CPI).

This implementation has poor cohesion (one class encapsulates everything), very strong coupling between the two classes, and adding another category of student (e.g., having a separate category for PhD students) will require making changes in existing code. Clearly, this poor design can be improved through refactoring.

As refactoring involves frequent testing, before refactoring, it is best to have an automated test script which can be used to test the changes as they are made. For this implementation, we have created a test script using JUnit (we will discuss this more later in the chapter). The test script essentially first creates a few students and sets suitable values (through a constructor), then invokes printReport() to print the report on a file, and then uses different assertions provided in JUnit to check if the output is as expected. (The test script is also available from the Web site.) This script will be executed every time a refactoring is done to check if anything is "broken." As refactoring should not change external behavior, it should be possible to use the earlier test scripts.

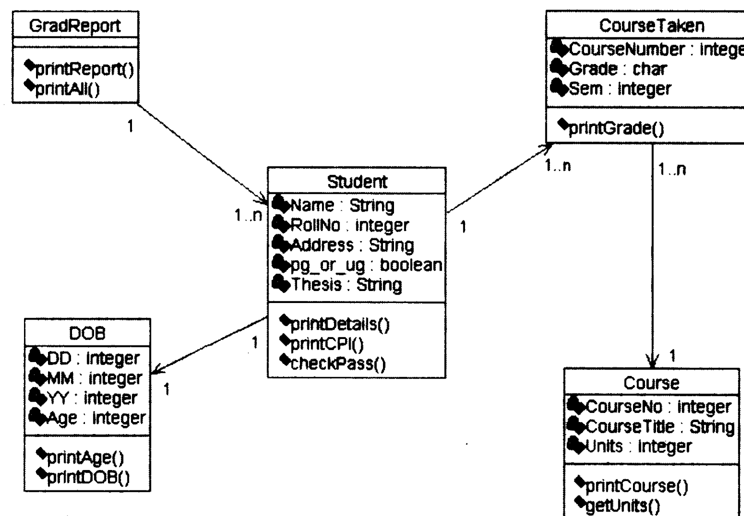

Figure 9.6: Class diagram after second refactoring.

To improve this design, we will perform a series of refactoring steps. In the first step, we improve the cohesion by creating a new class for Course, which contains all information about the course, and a class CourseTaken which contains information

related to a student taking a course (for example, the semester in which the course is taken and the grade the student gets). The responsibilities are also suitably shifted to these classes (and the constructor of Student also distributes the necessary construction activities among constructors of the different classes). The design after this refactoring is shown in 9.5.

With this refactoring, arrays in the earlier code have been converted into objects. Furthermore, redundancy in course information, which existed in earlier design (information about a course was replicated in each Student object that had that course) has also been eliminated.

The code after this refactoring can be tested using the test script for the original code. In other words, the earlier test script can be executed directly with this refactored code.

This design, though much improved, can still be improved. We can see that though date of birth of a student is information about a student, we can easily create an object to represent dates. This will be a more flexible and cohesive design. Also, the responsibility of printing different parts still rests within the main printReport() function, even though some of the information that is to be printed resides now with different objects. This increases coupling as the GradReport object will have to invoke methods on different objects to get the information it needs, and then print it. In this refactoring, functionality is distributed among objects such that the functionality resides where it belongs, that is, the functionality is performed in the object that has most of the information needed to perform the required function. This has been done with some printing functions as well as calculation of the grade point average. The design after this refactoring is shown in Figure 9.6. Again, as the external interface is preserved, the earlier test script can be used to execute this program and check that it passes.

In the final refactoring, we make use of inheritance. We note that in this design, coupling has been reduced and cohesion has been improved, the open-closed principle is still violated. If we were to add another class of student, then the code will have to be changed—we cannot handle it by extending the classes. This is because we are considering all students together and are separating the UG and PG students using a flag field. Using the power of inheritance, we can create a hierarchy in which we have different types of students as specializations of the base student class. This is what is done in this refactoring, and the final design is shown in 9.7. Now that we have PG as a separate class, as thesis is done only by PG students, thesis also has been made a separate class. Responsibilities have been suitably distributed. Due to the use of polymorphism, the flag variable (ug-or-pg) now disappears, and the conditionals using this flag have been replaced with suitable use of polymorphism.

Once again, the main interface remains the same and the code after this refactoring can be tested using the initial test script, thereby ensuring that whatever worked in

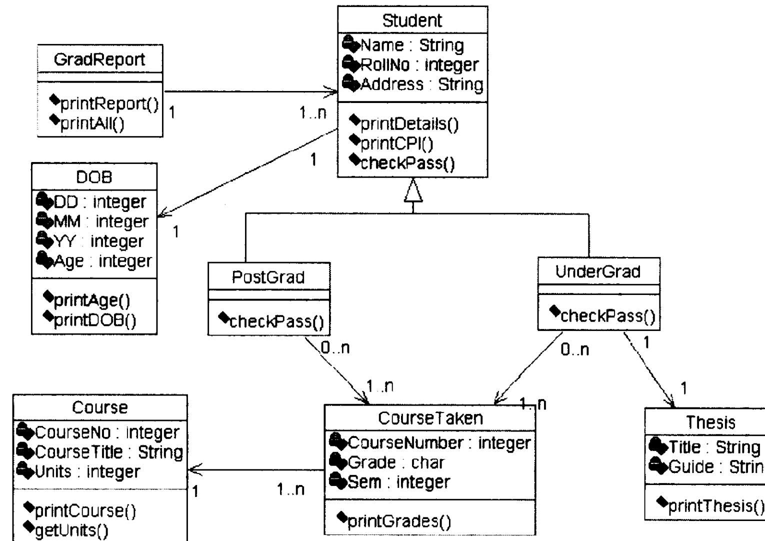

Figure 9.7: Class diagram after final refactoring.

the start (at least to the extent determined by the test script) continues to work after refactoring. For this example, the original code, code after each refactoring, and the Junit test script are all available from the Web site.

9.3.3 Bad Smells

We now discuss the signs in the code that can tell us that refactoring may be called for. These are sometimes called "bad smells" [65]. Basically, these are some easy to spot signs in the code that often indicate that some of the desirable design properties may be getting violated or that there is potential of improving the design. In other words, if you "smell" one of these "bad smells" it may be a sign that refactoring is needed. Of course, whether refactoring is indeed needed will have to be decided on a case-by-case basis by looking at the code and the opportunities that may exist for improving the code. Some of these bad smells from [65] are given here.

1. *Duplicate Code.* This is quite common. One reason for this is that some small functionality is being executed at multiple places (e.g., the age from date of birth may be computed in each place that needs the date). Another common reason is that when there are multiple subclasses of a class, then each subclass may end up doing a similar thing. Duplicate code means that if this logic or function has to be changed, it will have to be changed in all the places it exists, making changes much harder and costlier.

2. *Long Method.* If a method is large, it often represents the situation where it is trying to do too many things and therefore is not cohesive.

3. *Long Class.* Similarly, a large class may indicate that it is encapsulating multiple concepts, making the class not cohesive.

4. *Long Parameter List.* Complex interfaces are clearly not desirable—they make the code harder to understand. Often, the complexity is not intrinsic but a sign of improper design.

5. *Switch Statements.* In object-oriented programs, if the polymorphism is not being used properly, it is likely to result in a switch statement everywhere the behavior is to be different depending on the property. Presence of similar switch statements in different places is a sign that instead of using class hierarchy, switch statement is being used. Presence of switch statement makes it much harder to extend code—if a new category is to be added, all the switch statements will have to be modified.

6. *Speculative Generality.* Some class hierarchies may exist because the objects in subclasses seem to be different. However, if the behavior of objects of the different subclasses is the same, and there is no immediate reason to think that behaviors might change, then it is a case of unnecessary complexity.

7. *Too Much Communication Between Objects.* If methods in one class are making many calls to methods of another object to find out about its state, this is a sign of strong coupling. It is possible that this may be unnecessary and hence such situations should be examined for refactoring.

8. *Message Chaining.* One method calls another method, which simply passes this call to another object, and so on. This chain potentially results in unnecessary coupling.

These bad smells in general are indicative of a poor design. We can note that many of these smells existed in our example above.

9.3.4 Common Refactorings

Clearly there are unlimited possibilities of how code can be refactored to improve its design. A catalog of common refactorings, and steps for performing each of them is presented in [65]. New refactorings are continually being listed in www.refactoring.com. As discussed above, a refactoring should help make the code easier to understand and modify. To achieve this objective, many refactorings focus on improving the methods, classes, or class hierarchies. Here we discuss briefly some of the refactorings suggested in

[65] in each of these three groups. There are other refactorings that deal with reducing the message chains, which we do not discuss.

The general process of refactoring is the same for all of these—one refactoring step is performed, and then the new code is tested (with automated test scripts) to make sure that refactoring has not altered any behavior and old tests still pass. If multiple refactorings are to be applied, they should be applied one at a time, and a new one should be done only after the current one has been successfully tested.

Improving Methods

We have seen earlier that a method may not be cohesive and a method may perform many different functions. A main goal of refactoring to improve methods is to simplify them and make them more cohesive. The level of coupling by a method depends considerably on its interface and by simplifying the interface, the coupling can be reduced.

1. *Extract Method.* This refactoring is often done if a method is too long, indicating that it may not be cohesive and may be performing multiple functions. The objective is to have short methods whose signatures give a fairly accurate idea to the users about what the methods do. During this refactoring, a piece of code from a method is extracted out as a new method. The variables referred to in this code that are in the scope of original method become the parameters to the new method. Any variables declared in this code but used elsewhere will have to be defined in the original method. In the original method, the extracted code is replaced by an invocation to the new method. Sometimes, the new method may be a function returning a value. An illustration of this refactoring was given in the example above.

   Similarly, if there is a method that returns a value but also changes the state of some objects, then this method should be converted into two methods—one that returns the desired value, and the other to make the desired state change. Having a method that only returns a value and has no side effect generally has a strong functional cohesion. (This refactoring is called *separate query from modifier.*)

2. *Add/Remove Parameter.* If a method needs more information from its caller, perhaps because the scope of what the method has to do has expanded, then new parameters need to be added. However, this should be done only if the existing parameters cannot provide the information that is needed. The dual of this is parameter removal. Sometimes, for the sake of future extension or flexibility, information is asked for but is not used. For the sake of simplicity, parameters that are not being used should be removed. But if a class is part of a hierarchy, this has to be done carefully to make sure that hierarchy relationships are not disturbed. Or the change will have to be done at higher/lower levels of the hierarchy as well.

Improving Classes

Most refactorings under this category focus on improving cohesion of classes or reducing coupling between classes. Enhancing the cohesion of classes often results in moving fields and methods from one class to another such that logically connected data items and methods that access them are all encapsulated together in one object. Coupling reduction often requires changes in classes to reduce the degree of interaction between them.

1. *Move Method.* In this a method is moved from one class to another. This is a very important refactoring, and is generally done when a method in class A seems to be interacting too heavily with another object of class B, indicating that perhaps the natural home for the method is class B and not class A. Often it is not initially clear where a method may belong, and the designer may assign it to class A. However, later, if it is seen that the natural home is class B, then it should be moved.

   Initially, it is better to leave the source method as a delegating method which just invokes the new method. This way, the change is limited only to the two methods. However, whenever possible, the references or calls to the methods should be redirected to the new method and the old method should be eliminated. If the original method was not a private method, then this will imply that all the classes that used the method will have to be changed.

2. *Move Field.* If a field in class A is being used more often by methods in class B, then the field should be moved to class B. This will reduce the coupling between A and B, and enhance the encapsulation of both the classes. This refactoring is similar to the one above—in Move Method the behavior is being reassigned and in this one state is being reassigned. Assuming that the field was private, after moving the field from class A to B, all reference to the field in methods in A will have to be changed to suitable method calls to class B for getting the state.

3. *Extract Class.* Often a designer starts with a class and as the need for new features arise, the classes are extended to do more, sometimes resulting in large classes that do not have clear abstraction and are holding too many responsibilities. If there is a large class that seems to be holding multiple responsibilities or encapsulating more than one concept, then this refactoring is applied. A new class is created and the relevant fields and methods are moved to the new class. The refactoring is justified if the new classes have crisper abstraction than the large class, and the responsibilities of both can be clearly and succinctly stated. The class extraction should not result in too much coupling between the two classes, which will indicate an artificial partitioning. If the large class was holding multiple responsibilities, the chances are that different subsets of its methods are primarily accessing different subsets of its state.

One way to perform this refactoring is to first create the new class and create a link from the old class to the new one. Then the move field and move method refactorings can be repeatedly applied to move the fields and methods that belong to the new class.

4. *Replace Data Value with Object.* This is similar to the Extract Class refactoring. Often some data items are treated as fields of the class initially. As development proceeds, these data items become semantically richer with more operations being performed on them. Examples of such data items are date, telephone numbers, social security number, address etc. If multiple operations are being performed within the class on these data items, then it may make sense to convert the data into an object.

Improve Hierarchies

Class hierarchy is a key object oriented concept that is the foundation for the open-closed principle. In order to support this principle, it is imperative that polymorphism be used effectively. The goal of refactorings here is to leverage polymorphism to make classes more amenable to easy extension later, and to use polymorphism to create designs that more naturally represent the problem.

1. *Replace Conditional with Polymorphism.* If we have a class in which the behavior of some methods depends on value of some type-code, we essentially have a situation where a traditional, function-oriented approach is still being used. Polymorphism can, and should, be used to capture the situation more naturally. Presence of case statement (or equivalent) within a method, or some type codes declared in the class, are good indicators that this refactoring may be needed. An example of this was given earlier, when in the method printReport() of GradReport class, which has to deal with behavior of PG and UG students. In refactoring, a class hierarchy was created where PG and UG are modeled into two different classes. These objects have some common functionality inherited from their base class Student, apart from having their specialized methods. So the object of base class can now point dynamically to one of the derived classes which got rid of the case statement. The difference in behavior was captured by the different sub classes and hence there is no need of an explicit switch statement.

2. *Pull up Field/Method.* An important factor in having a good hierarchy is to have common elements belonging to parent class while the variable elements belonging to the sub classes. Consequently, when there is a situation that multiple subclasses have the same field, the field should be pulled up to the superclass. Similarly, if there are methods in subclasses that are producing identical results (perhaps even with different code/algorithm) we have a situation where functionality is being

duplicated. And duplicate code is one of the key factors that makes making changes much harder. Hence, such a situation exists, the structure is improved if the method is moved up to the superclass. If the subclasses have constructors which are similar, then they can also be pulled up into the superclass and be called from the subclass constructor.

The inverse of pull up is push down, giving us push down field/method refactorings. If a superclass contains a field that is used only by some subclass, it is best to push that field down to the class that uses it. Similarly, if there is some method in the superclass that is relevant only to some subclass, then it should be moved down to the subclass.

9.4 Verification

Once a programmer has written the code for a module, it has to be verified before it is used by others. So far we have assumed that testing is the means by which this verification is done. Though testing is the most common method of verification, there are other effective techniques also. Here we discuss a few common ones. It should be pointed out that by verification we do not mean proving correctness of programs, which for our purposes is only *one* method for program verification.

Here we will focus on techniques that are now widely used in practice—inspections (including code reading), unit testing, and program checking. We will also discuss a formal verification approach, though formal verification is less widely used and is applied mostly in special situations.

Though we are focusing on verifying individual programs written by programmers, some of the techniques like program checking are applicable at the complete system level also.

9.4.1 Code Inspections

Inspection, which is a general verification approach that can be applied to any document, has been widely used for detecting defects. It was started for detecting defects in the code, and was later applied for design, requirements, plans, etc. The general inspection process was discussed earlier, and for code inspection also it remains the same.

Code inspections are usually held after code has been successfully compiled and other forms of static tools have been applied. The main motivation for this is to save human time and effort, which would otherwise be spent detecting errors that a compiler or static analyzer can detect.

The documentation to be distributed to the inspection team members includes the code to be reviewed and the design document. The team for code inspection should include the programmer, the designer, and the tester.

The aim of code inspections is to detect defects in code. In addition to defects, there are quality issues which code inspections usually look for, like efficiency, compliance to coding standards, etc. Often the type of defects the code inspection should focus on is contained in a checklist that is provided to the inspectors. Some of the items that can be included in a checklist for code reviews are [52]:

A Sample Checklist:

- Do data definitions exploit the typing capabilities of the language?
- Do all the pointers point to some object? (Are there any "dangling pointers"?)
- Are the pointers set to NULL where needed?
- Are pointers being checked for NULL when being used?
- Are all the array indexes within bound?
- Are indexes properly initialized?
- Are all the branch conditions correct (not too weak, not too strong)?
- Will a loop always terminate (no infinite loops)?
- Is the loop termination condition correct?
- Is the number of loop executions "off by one"?
- Where applicable, are the divisors tested for zero?
- Are imported data tested for validity?
- Do actual and formal interface parameters match?
- Are all variables used? Are all output variables assigned?
- Can statements placed in the loop be placed outside the loop?
- Are the labels unreferenced?
- Will the requirements of execution time be met?
- Are the local coding standards met?

Inspection are very effective for detecting defects and are widely used in many commercial organizations. However, inspections also tends to be very expensive as it uses time of many people. Consequently, for some code segments the cost may not be justified. In these situations, instead of a group inspection, review by one person can be

performed. One approach for doing this is to have the person inspecting the code apply some structured code reading technique, which we briefly discuss now.

Code reading involves careful reading of the code by the reviewer to detect any discrepancies between the design specifications and the actual implementation. It involves determining the abstraction of a module and then comparing it with its specifications. The process is the reverse of design. In design, we start from an abstraction and move toward more details. In code reading we start from the details of a program and move toward an abstract description.

The process of code reading is best done by reading the code inside-out, starting with the innermost structure of the module. First determine its abstract behavior and specify the abstraction. Then the higher-level structure is considered, with the inner structure replaced by its abstraction. This process is continued until we reach the module or program being read. At that time the abstract behavior of the program/module will be known, which can then be compared to the specifications to determine any discrepancies.

Code reading is very useful and can detect errors often not revealed by testing. Reading in the manner of stepwise abstraction also forces the programmer to code in a manner conducive to this process, which leads to well-structured programs. Code reading is sometimes called *desk review*.

9.4.2 Static Analysis

There are many techniques for verification now available that are not testing-based, but directly check the programs through the aid of analysis tools. This general area is called program checking. Three forms of checking are becoming popular—model checking, dynamic analysis, and static analysis. (Program verification can also be treated as a form of program checking, but is generally not performed through tools. We discuss it separately later in the section.)

In model checking, an abstract model of the program being verified is first constructed. The model captures those aspects that affect the properties that are to be checked. The desired properties are also specified and a model checker checks whether the model satisfies the stated properties. A discussion of model checking is available in [56, 42]. In dynamic analysis, the program is instrumented and then executed with some data. The value of variables, branches taken, etc. are recorded during the execution. Using the data recorded, it is evaluated if the program behavior is consistent with some of the dynamic properties. A discussion of dynamic analysis is available in [117, 3]. Perhaps the most widely used program checking technique is static analysis, which is becoming increasingly popular with more tools becoming available. In this section we focus primarily on static analysis.

Analysis of programs by methodically analyzing the program text is called *static analysis*. Static analysis is usually performed mechanically by the aid of software tools. During static analysis the program itself is not executed, but the program text is the

input to the tools. The aim of static analysis is to detect errors or potential errors in the code and to generate information that can be useful in debugging. (Static analyzers can also generate information for documentation, but we will not discuss this aspect.)

Many compilers perform some limited static analysis. However, the analysis performed by compilers focuses around code generation and not defect detection. Static analysis tools, on the other hand, explicitly focus on detecting errors. Two approaches are possible. The first is to detect patterns in code that are "unusual" or "undesirable" and which are likely to represent defects. The other is to directly look for defects in the code, that is, look for those conditions that can cause programs to fail when executing.

In either case, a static analyzer, as it is trying to identify defects (i.e. which can cause failures on execution) without running the code but only by analyzing the code, sometimes identifies situations as errors which are not actually errors (i.e. false positives), and sometimes fails to identify some errors. These limitations of a static analyzer is characterized by its *soundness* and *completeness.* Soundness captures the occurrence of false positives in the errors the static analyzer identifies, and completeness characterizes how many of the existing errors are missed by the static analyzer. As full soundness and completeness is not possible, the goal is to have static analyzers be as sound and as complete as possible. Usually there is a trade off involved—a higher level of completeness often implies less soundness (i.e., more false positives). Due to imperfect soundness, the errors identified by static analyzers are actually "warnings"—the program possibly has a defect, but there is a possibility that the warning may not be a defect.

The first form of static analysis is looking for unusual patterns in code. This is often done through data flow analysis and control flow analysis. One of the early approaches focusing of data flow anamolies is described in [63]. Here, our discussion is based on checkers described in [150], which identify redundancies in the programs. These redundancies usually go undetected by the compiler, and often represent errors caused due to carelessness in typing, lack of clear understanding of the logic, or some other reason. At the very least, presence of such redundancies implies poor coding. Hence, if a program has these redundancies it is a cause of concern, and their presence should be carefully examined. Some of the redundancies that the checkers identify are:

- Idempotent operations
- Assignments that were never read
- Dead code
- Conditional branches that were never taken

Idempotent operations occur in situations like when a variable is assigned to itself, divided by itself, or performs a boolean operation with itself. Redundant assignments occur when a variable is assigned some value but the variable is not used after the assignment, that is, either the function exits or a new assignment is done without using

```
/* idempotent operation */
for (i=0; i< size; i++) {
     if (pv[i]!= -1 && pv[i] >= val)
     pv[i] = pv[i]++; /* error */
}

/* Redundant assignment */
do {
     ...
     if (signal\_pending(current))
     { err = - ERRSTARTSYS; break; }
     ...
} while (condition);
return 0; /*value of err lost*/

/* Dead code */
for (c1; c2; c3) {
...
if (C) {
     ...
     break; }
else {
     ...
     break; }
stmt;  /*this is unreachable*/

/* Unnecessary check */
if (!(error && ... && ...))
{   ...
     return -1; }
if (error) /*redundant check*/
     { ... }
}
```

Figure 9.8: Examples of redundant operations.

the variable. Dead code occurs when there is a piece of code that cannot be reached in any path and consequently will never be executed. Redundant conditionals occur if a branching construct contains a condition that is always true or false, and hence is redundant. All these situations represent redundancies in programs that should normally not occur in well thought out programs. Hence, they are candidates for presence of errors.

Some examples of errors identified by these checks will illustrate the use of techniques. Small program fragments from large public domain software systems which contained these redundancies are shown in Figure 9.8 [150]. In these examples the presence of these redundancies actually represents some type of error in the program.

These checkers are efficient and can be applied on large code bases. Experiments on many widely used software systems have shown that the warnings generated by the static analyzer has reasonable levels of "false positives" (about 20% to 50% of the warnings are false positives). Experiments also showed that the presence of these redundancies correlate highly with actual errors in programs.

The second approach for static analysis is to directly look for errors in the programs—bugs that can cause failures when the programs execute. These approaches focus on some types of defects that are otherwise hard to identify or test. Many tools of this type are commercially available or have been developed in-house by large software organizations. Here we base our discussion on the tool called PREfix [28], which has been used on some very large software systems and in some large commercial software companies. As they directly look for errors, the level of false positives generated by this tool tends to be low. Some of the errors PREfix identifies are:

- Using uninitialized memory
- Dereferencing uninitialized pointer
- Dereferencing NULL pointer
- Dereferencing invalid pointer
- Dereferencing or returning pointer to freed memory
- Leaking memory or some other resource like a file
- Returning pointer to local stack variable
- Divide by zero

As we can see, these are all situations that can cause failure of the software during execution. Also, as we have discussed earlier, some of these errors are made commonly by programmers and are often hard to detect through testing. In other words, many of these errors occur commonly and are hard to detect, but can be detected easily and cheaply by the use of this tool.

To identify these errors, PREfix simulates the execution of functions by tracing some distinct paths in the function. As the number of paths can be infinite, a maximum limit is set on the number of paths that will be simulated. As it turns out, most of the errors get detected within a limit of about 100 paths. During simulation of the execution, it keeps track of the memory state, which it also examines at the end of the path and reports the memory problems it identifies. (As we can see from the list above, the focus is quite heavily on memory related errors.)

For complete programs, it first simulates the lowest level functions in the call graph, and then moves up the graph. For a called function, a model is built by simulation,

```
1.  #include <stdlib.h>
2.  #include <stdio.h>

3.  char *f(int size)
4.  {
5.      char *result;

6.      if (size>0)
7.          result = (char *)malloc(size);
8.      if (size==1)
9.          return NULL;
10.     result[0] = 0;
11.     return result;
12. }
```

Figure 9.9: An example program.

which is then used in simulation of the called function. The model of a function consists of the list of external variables that affect the function (parameters, global variables, etc.) or that the function affects (return values, global variables, etc.), and a set of possible outcomes. Each outcome consists of a guard which specifies the pre-condition for this outcome, constraints, and the result (which is essentially the post-conidition). By simulating called functions and using their outcomes in the simulation of a called function allows the tool to identify inter-function problems—their experiments showed that more than 90% of the errors fall in this category where more than one function is involved. Details of how the analysis is done are given in the paper [28]. An example of the types of errors identified will illustrate what the tool does. Consider the program given in Figure 9.9 [28]. The tool will generate three warnings for this program:

```
8: leaking memory (path: 5 6 7 8)
9: dereferencing uninitialized pointer 'result'
   (path: 5 7 9)
9: dereferencing NULL pointer 'result'
   (path: 5 6 7 9)
```

The first warning catches the error that if size is 1, then the allocated memory is not freed, and hence we have a memory leak. The second warning catches the error that if size is less than or equal to 0, then line 6 will not be executed, and hence result is not defined and we access an uninitialized pointer. If this path is followed in an execution of the program, a runtime error will be generated at line 9. Similarly, if malloc() cannot allocate memory and returns a NULL pointer in line 6, then there will be a runtime error of trying to dereference a NULL pointer at line 9.

All these are runtime errors that are detected not by executing the program but by analyzing the program text. Besides the nature of the error found, the tool gives the path in whose execution the tool found the error—this helps the programmer in understanding under which situation the error occurs. The tool provides a lot more information to help the programmer clearly identify the error.

As static analysis is performed with the help of software tools, it is a very cost-effective way of discovering errors. An added advantage of static analysis is that it detects the errors directly and not just the presence of errors, as is the case with testing. Consequently, little debugging is needed after the presence of error is detected. The main issue with using these tools is the presence of "false positives" in the warnings the tool generates. The presence of false positives means that a programmer has to also examine the false positives and then discard them, leading to wastage of effort. More importantly, they cause a doubt in the minds of the programmer on the warnings which can lead to even correct errors being discarded as false positives. Still, the use of static analysis is increasing in commercial setups as they provide a cost effective and scalable technique of detecting errors in the code which are often hard to detect through testing.

The general area of program checking is an active area of research. There are many commercial and public domain tools available for performing different types of analysis.

9.4.3 Proving Correctness

Many techniques for verification aim to reveal errors in the programs, because the ultimate goal is to make programs correct by removing the errors. In proof of correctness, the aim is to prove a program correct. So, correctness is directly established, unlike the other techniques in which correctness is never really established but is implied (and hoped) by the absence of detection of any errors. Proofs are perhaps more valuable during program construction, rather than after the program has been constructed. Proving while developing a program may result in more reliable programs that can be proved more easily. Proving a program not constructed with formal verification in mind can be quite difficult.

Any proof technique must begin with a formal specification of the program. No formal proof can be provided if what we have to prove is not stated or is stated informally in an imprecise manner. So, first we have to state formally what the program is supposed to do. A program will usually not operate on an arbitrary set of input data and may produce valid results only for some range of inputs. Hence, it is often not sufficient merely to state the goal of the program, but we should also state the input conditions in which the program is to be invoked and for which the program is expected to produce valid results. The assertion about the expected final state of a program is called the *post-condition* of that program, and the assertion about the input condition is called the *pre-condition* of the program. Often, determining the pre-condition for which the post-condition will be satisfied is the goal of proof. Here we will briefly describe a

technique for proving correctness called the *axiomatic method*, which was proposed by Hoare [86]. It is often also called the *Floyd-Hoare proof method*, as it is based on Floyd's inductive assertion technique.

The Axiomatic Approach

In principle, all the properties of a program can be determined statically from the text of the program, without actually executing the program. The first requirement in reasoning about programs is to state formally the properties of the elementary operations and statements that the program uses. In the axiomatic model of Hoare [86], the goal is to take the program and construct a sequence of assertions, each of which can be inferred from previously proved assertions and the rules and axioms about the statements and operations in the program. For this, we need a mathematical model of a program and all the constructs in the programming language. Using Hoare's notation, the basic assertion about a program segment is of the form:

$$P\{S\}Q.$$

The interpretation of this is that if assertion P is true before executing S, then assertion Q will be true after executing S, if the execution of S terminates. Assertion P is the pre-condition of the program and Q is the post-condition. These assertions are about the values taken by the variables in the program before and after its execution. The assertions generally do not specify a particular value for the variables, but they specify the general properties of the values and the relationships among them.

To prove a theorem of the form $P\{S\}Q$, we need some rules and axioms about the programming language in which the program segment S is written. Here we consider a simple programming language, which deals only with integers and has the following types of statements: (1) assignment, (2) conditional statement, and (3) an iterative statement. A program is considered a sequence of statements. We will now discuss the rules and axioms for these statements so that we can combine them to prove the correctness of programs.

Axiom of Assignment: Assignments are central to procedural languages. In our language no state change can be accomplished without the assignment statement. The axiom of assignment is also central to the axiomatic approach. In fact, only for the assignment statement do we have an independent axiom; for the rest of the statements we have rules. Consider the assignment statement of the form

$$x := f$$

where x is an identifier and f is an expression in the programming language without any side effects. Any assertion that is true about x after the assignment must be true of the expression f before the assignment. In other words, because after the assignment the variable x contains the value computed by the expression f, if a condition is true

after the assignment is made, then the condition obtained by replacing x by f must be true before the assignment. This is the essence of the axiom of assignment. The axiom is stated next:

$$P_f^x\{x := f\}P$$

P is the post-condition of the program segment containing only the assignment statement. The pre-condition is P_f^x, which is an assertion obtained by substituting f for all occurrences of x in the assertion P. In other words, if P_f^x is true before the assignment statement, P will be true after the assignment.

This is the only axiom we have in Hoare's axiomatic model besides the standard axioms about the mathematical operators used in the language (such as commutativity and associativity of the + operator). The reason that we have only one axiom for the assignment statement is that this is the only statement in our language that has any effect on the state of the system, and we need an axiom to define what the effect of such a statement is. The other language constructs, like alternation and iteration, are for flow control, to determine which assignment statements will be executed. For such statements rules of inference are provided.

Rule of Composition: Let us first consider the rule for sequential composition, where two statements S1 and S2 are executed in sequence. This rule is called *rule of composition*, and is shown next:

$$\frac{P\{S1\}Q, Q\{S2\}R}{P\{S1; S2\}R}$$

The explanation of this notation is that if what is stated in the numerator can be proved, the denominator can be inferred. Using this rule, if we can prove P{S1}Q and Q{S2}R, we can claim that if before execution the pre-condition P holds, then after execution of the program segment S1;S2 the post-condition R will hold. In other words, to prove P{S1;S2}R, we have to find some Q and prove that P{S1}Q and Q{S2}R. This rule is dividing the problem of determining the semantics of a sequence of statements into determining the semantics of individual statements. In other words, from the proofs of simple statements, proofs of programs (i.e., sequence of statements) will be constructed. Note that the rule handles a strict sequence of statements only (recall the earlier discussion on structured programming).

Rule for Alternate Statement: Let us now consider the rules for an `if` statement. For formal verification, the entire `if` statement is treated as one construct, the semantics of which have to be determined. This is the way in which other structured statements are also handled. There are two types of `if` statement, one with an `else` clause and one without. The rules for both are given next:

$$\frac{P \wedge B\{S\}Q, P \wedge \sim B \Rightarrow Q}{P\ \{\text{if B then S}\}Q}$$

$$\frac{P \wedge B\{S1\}Q, P \wedge \sim B\{S2\}Q}{P\{\text{if B then S1 else S2}\}Q}$$

Let us consider the if-then-else statement. We want to prove a post-condition for this statement. However, depending on the evaluation of B, two different statements can be executed. In both cases the post-condition must be satisfied. Hence if we can show that starting in the state where $P \wedge B$ is true and executing S1 or starting in a state where $P \wedge \sim B$ is true and executing the statement S2, both lead to the post-condition Q, then the following can be inferred: if the if-then-else statement is executed with pre-condition P, the post-condition Q will hold after execution of the statement. Similarly, for the if-then statement, if B is true then S is executed; otherwise the control goes straight to the end of the statement. Hence, if we can show that starting from a state where $P \wedge B$ is true and executing S leads to a state where Q is true and before the `if` statement if $P \wedge \sim B$ implies Q, then we can say that starting from P before the `if` statement we will always reach a state in which Q is true.

Rules of Consequence: To be able to prove new theorems from the ones we have already proved using the axioms, we require some rules of inference. The simplest inference rule is that if the execution of a program ensures that an assertion Q is true after execution, then it also ensures that every assertion logically implied by Q is also true after execution. Similarly, if a pre-condition ensures that a post-condition is true after execution of a program, then every condition that logically implies the pre-condition will also ensure that the post-condition holds after execution of the program. These are called *rules of consequence*, and they are formally stated here:

$$\frac{P\{S\}R,\ R \Rightarrow Q}{P\{S\}Q}$$

$$\frac{P \Rightarrow R,\ R\{S\}Q}{P\{S\}Q}$$

Rule of Iteration: Now let us consider iteration. Loops are the trickiest construct when dealing with program proofs. We will consider only the `while` loop of the form `while B do S`. We have to determine the semantics of the whole construct.

In executing this loop, first the condition B is checked. If B is false, S is not executed and the loop terminates. If B is true, S is executed and B is tested again. This is repeated until B evaluates to false. We would like to be able to make an assertion that will be true when the loop terminates. Let this assertion be P. As we do not know how many times the loop will be executed, it is easier to have an assertion that will hold true irrespective of how many times the loop body is executed. In that case P will hold true after every execution of statement S, and will be true before every execution of S, because the condition that holds true after an execution of S will be the condition for the next execution of S (if S is executed again). Furthermore, we know that the condition B is false when the loop terminates and is true whenever S is executed. These properties have been used in the rule for iteration:

$$\frac{P \wedge B\{S\}P}{P\{\text{while B do S}\}P \wedge \sim B}$$

```
(* Remainder of x/y *)
1. begin
2.     q := 0;
3.     r := x;
4.     while r ≥ y do
5.     begin
6.         r := r - y ;
7.         q := q + 1 ;
8.     end;
9.end.
```

Figure 9.10: Program to determine the remainder.

As the condition P is unchanging with the execution of the statements in the loop body, it is called the *loop invariant.* Finding loop invariants is the thorniest problem in constructing proofs of correctness. One method for getting the loop invariant that often works is to extract ~B from the post-condition of the loop and try the remaining assertion as the loop invariant. Another method is to try replacing the variable that binds the loop execution with the loop counter. Thus if the loop has a counter i, which goes from 0 to n, and if the post-condition of the loop contains n, then replace n by i and try the assertion as a loop invariant.

An Example

Although in a theorem of the form $P\{S\}Q$, we say that if P is true at the start and the execution of S terminates, Q will be true after executing S, to prove a theorem of this sort we work backwards. That is, we do not start with the pre-condition; we work our way to the end of the program to determine the post-condition. Instead we start with the post-condition and work our way back to the start of the program, and determine the pre-condition. We use the axiom of assignment and other rules to determine the pre-condition of a statement for a given post-condition. If the pre-condition we obtain by doing this is implied by P, then by rules of consequence we can say that P{S}Q is a theorem. Let us consider a simple example of determining the remainder in integer division, by repeated subtraction. The program is shown in Figure 9.10.

The pre-condition and post-condition of this program are given as

$$P = \{x \geq 0 \wedge y > 0\}$$

$$Q = \{x = qy + r \wedge 0 \leq r < y\}$$

We have to prove that P {Program} Q is a theorem. We start with Q. The first statement before the end of the program is the loop. We invent the loop invariant by removing ~B from the Q, which is also the output assertion of the loop. For this we factor Q into a form like $I \wedge \sim$B, then choose I as the invariant. For this program we have $\sim B = \{r < y\}$, and $Q = \{x = qy + r \wedge 0 \leq r \wedge r < y\}$, hence our trial invariant I is $\{x = qy + r \wedge 0 \leq r\}$.

Let us now see if this invariant is appropriate for this loop, that is, starting with this, we get a pre-condition of the form $I \wedge B$. Starting with I, we use the assignment axiom and the pre-condition for statement 7 is

$$x = (q+1)y + r \wedge 0 \leq r\{q := q+1\}I$$

Using the assignment axiom for statement 6, we get the pre-condition for 6 as

$$x = (q+1)y + (r - y) \wedge 0 \leq (r - y).$$

which is the same as $x = qy + r \wedge y \leq r$. Using the rule of composition (for statements 6 and 7), we can say

$$x = qy + r \wedge y \leq r\{r := r - y; q := q + 1\}I.$$

Because $x = qy + r \wedge y \leq r \Rightarrow I \wedge B$, by rule of consequence and the rule for the `while` loop, we have

$$I\{\text{while loop in program}\}I \wedge \sim(r \geq y)$$

where I is $x = qy + r \wedge 0 \leq r$.

Now let us consider the statements before the loop (i.e., statements 2 and 3). The post-condition for these statements is I. Using the axiom of assignment, we first replace r with x, and then we replace q with 0 to get

$$(x = x \wedge 0 \leq x) \Rightarrow (0 \leq x).$$

By composing these statements with the `while` statement, we get

$$0 \leq x\{\text{the entire program}\}I \wedge \sim B$$

Because, $(I \wedge \sim B)$ is the post-condition Q of the program and 0≤x is the pre-condition, we have proved the program to be correct.

Discussion

In the axiomatic method, to prove P{S}Q, we assume that S will terminate. So, by proving that the program will produce the desired post-condition using the axiomatic method, we are essentially saying that *if* the program terminates, it will provide the desired post-condition. The axiomatic proof technique cannot prove whether or not a

program terminates. For this reason, the proof using the axiomatic technique is called the proof of *partial correctness.*

This is in contrast to the proof of *total correctness*, where termination of a program is also proved. Termination of programs is of considerable interest for obvious reason of avoiding infinite loops. With the axiomatic method, additional techniques have to be used to prove termination. One common method is to define a well-ordered set that has a smallest member and then add an expression to the assertions that produces a value in the set. If after an execution of the loop body, it can be shown that the value of the expression is less than it was on the entry, then the loop must terminate. There are other methods of proving correctness that aim to prove total correctness.

Proofs of correctness have obvious theoretical appeal and a considerable body of literature exists in the area. Despite this, the practical use of these formal methods of verification has been limited. In the software development industry proving correctness is not generally used as a means of verification. Their use, at best, is limited to proving correctness of some critical modules.

There are many reasons for the lack of general use of formal verification. Constructing proofs is quite hard, and even for relatively modest problems, proofs can be quite large and difficult to comprehend. As much of the work must be done manually (even if theorem provers are available), the techniques are open to clerical errors. In addition, the proof methods are usually limited to proving correctness of single modules. When procedures and functions are used, constructing proofs of correctness becomes extremely hard. In essence, the technique of proving correctness does not scale up very well to large programs. Despite these shortcomings, proof techniques offer an attractive formal means for verification and hold promise for the future.

9.4.4 Unit Testing

Unit testing is another approach for verifying the code that a programmer is written. Unit testing is like regular testing where programs are executed with some test cases except that the focus is on testing smaller programs or modules called units. In the programming processes we discussed earlier, the testing was essentially unit testing. A unit may be a function, a small collection or functions, a class, or a small collection of classes. Most often, it is the unit a programmer is writing code for, and hence unit testing is most often done by a programmer to test the code that he or she has written. Testing, however, is a general technique that can also be used for validating complete systems. We will discuss testing in more detail in the next chapter.

Testing of modules or software systems is a difficult and challenging task. Selection of test cases is a key issue in any form of testing. We will discuss the problem of test case selection in detail in the next chapter when we discuss testing. For now, it suffices that during unit testing the tester, who is generally the programmer, will execute the unit for a variety of test cases and study the actual behavior of the units being tested for these test cases. Based on the behavior, the tester decides whether the unit is

working correctly or not. If the behavior is not as expected for some test case, then the programmer finds the defect in the program (an activity called *debugging*), and fixes it. After removing the defect, the programmer will generally execute the test case that caused the unit to fail again to ensure that the fixing has indeed made the unit behave correctly.

For a functional unit, unit testing will involve testing the function with different test data as input. In this, the tester will select different types of test data to exercise the function. Typically, the test data will include some data representing the normal case, that is, the one that is most likely to occur. In addition, test data will be selected for special cases which must be dealt with by the program and which might result in special or exceptional result.

An issue with unit testing is that as the unit being tested is not a complete system but just a part, it is not executable by itself. Furthermore, in its execution it may use other modules that have not been developed yet. Due to this, unit testing often requires drivers or stubs to be written. Drivers play the role of the "calling" module and are often responsible for getting the test data, executing the unit with the test data, and then reporting the result. Stubs are essentially "dummy" modules that are used in place of the actual module to facilitate unit testing. So, if a module M uses services from another module M' that has not yet been developed, then for unit testing M, some stub for M' will have to be written so M can invoke the services in some manner on M' so that unit testing can proceed. The need for stubs can be avoided, if coding and testing proceeds in a bottom-up manner—the modules at lower levels are coded and tested first such that when modules at higher levels of hierarchy are tested, the code for lower level modules is already available.

If incremental coding is practiced, as discussed above, then unit testing needs to be performed every time the programmer adds some code. Clearly, for this, automated scripts for unit testing are essential. With automated scripts, whether the programs pass the unit tests or not can be determined simply by executing a script. For incremental testing it is desirable that the programmer develops this unit testing script and keeps enhancing it with additional test cases as the code evolves. That is, instead of executing the unit by executing it and manually inputting the test data, it is better if execution of the unit with the chosen test data is all programmed. Then this program can be executed every time testing needs to be done. Some tools are available to facilitate this.

In object-oriented programs, the unit to be tested is usually an object of a class. Testing of objects can be defined as the process of exercising the routines provided by an object with the goal of uncovering errors in the implementation of the routines or state of the object or both [137]. For an object, we can test a method using approaches for testing functions, but we cannot test the object using these approaches, as the issue of state comes in. To test an object, we also have to test the interaction between the methods provided on the object.

State-based testing is a technique that can be used for unit testing an object. In the simplest form, a method is tested in all possible states that the object can assume, and after each invocation the resulting state is checked to see whether or not the method takes the object under test to the expected state. For state-based testing to be practical, the set of states in which a method is tested has to be limited. State modeling of classes can help here [24, 64], or the tester can determine the important states of the object. Once the different object states are decided, then a method is tested in all those states that form valid input for it. We will discuss selection of test cases based on a state model in the next chapter.

To test a class, the programmer needs to create an object of that class, take the object to a particular state, invoke a method on it, and then check whether the state of the object is as expected. This sequence has to be executed many times for a method, and has to be performed for all the methods. All this is facilitated if we use frameworks like the Junit (www.junit.org). Though Junit itself is for Java, similar frameworks have been developed for other languages like C++ and C#. Here we briefly describe how Junit can be used for testing a class and give an example.

For testing of a class CUT (class under test) with Junit, the tester has to create another class which inherits from Junit (e.g., `class CUTtest extends Junit`). The Junit framework has to be imported by this class. This class is the driver for testing CUT. It must have a constructor in which the objects that are needed for the test cases are created; a setUp() method which is typically used for creating any objects and setting up values before executing a test case; a suite(), and a main () that executes the suite() using a TestRunner provided by Junit. Besides these methods, all other methods are actually test cases.

Most of these methods are often named `testxxxx()`. Such a method typically focuses on testing a method under some state (typically the name of the method and/or the state is contained in xxx). This method first sets up the state if not already setup (by setup()), and then executes the method to be tested. To check the results, Junit provides two special methods AssertTrue(boolean_expression) and AssertFalse(boolean_expression). By using functions and having a logical expression on the state of the object, the tester can test if the state is correct or not. If all the assertions in all the methods succeed, then Junit declares that the test has passed. If any assert statements fail, Junit declares that testing has failed and specifies the assertion that has failed.

To get an idea of how it works, consider the testing of a class Matrix.java, which provides standard operations on matrices. The main attributes of the class and the main methods are given in Figure 9.11.

For unit testing the Matrix class, clearly we need to test standard operations like creation of a matrix, setting of values, etc. We also need to test whether the operations like add, subtract, multiply, determinant are performing as expected. Each test case we want to execute is programmed by setting the values and then performing the operation.

```
class Matrix {
     private double [][] matrix;  //Matrix elements
     private int row, col;      //Order of Matrix

     public Matrix(); // Constructor
     public Matrix(int i,int j); // Sets #rows and #cols
     public Matrix(int i,int j,double[][] a); // Sets from 2D array
     public Matrix(Matrix a); //Constructs matrix from another
     public void read(); //read elts from console and set up matrix
     public void setElement(int i,int j,double value); // set elt i,j
     public int noOfRows(); // returns no of rows
     public int noOfCols(); // returns no of cols
     public Matrix add(Matrix a); // add a to matrix
     public Matrix sub(Matrix a); // subtracts a from matrix
     public Matrix mul(Matrix b); // multiplies b to matrix
     public Matrix transpose(); // transposes the matrix
     public Matrix minor(int a, int b); // returns a x b sub-matrix
     public double determinant(); // determinant of the matrix
     public Matrix inverse() throws Exception; // inverse of the matrix
     public void print(); // prints matrix on console
     public boolean equals(Matrix m); // checks for equality with m
}
```

Figure 9.11: Class Matrix.java

The result of the operation is checked through the assert statements. For example, for testing add(), we create a method testAdd() in which a matrix is added to another. The correct result is stored apriori in another matrix. After addition, it is checked if the result obtained by performing add() is equal to the correct result. The method for this is shown in Figure 9.12. The programmer may want to perform more tests for add(), for which more test methods will be needed. Similarly, methods are written for testing other methods. Some of these tests are also shown in Figure 9.12. The complete script has over 30 assertions spread over more than 20 test methods. The complete code for classes Matrix.java and MatrixTest.java can be found on the book's Web site.

As we can see, Junit encourages automated testing. Not only is the execution of test cases automated, the checking of the results is also automated. This makes running tests fully automatic. By building testing scripts, and continuously updating them as the class evolves, we always have a test script which can be run quickly. So, whenever we make any changes to the code, we can quickly check if the past test cases are running on the click of a button. This becomes almost essential if incremental coding or test driven development (discussed earlier in the chapter) is to be practiced.

9.4.5 Combining Different Techniques

After discussing various techniques for verification it is natural to ask how these techniques compare with each other, and how they should be combined for applying it on

```
import junit.framework.*;
public class MatrixTest extends TestCase {

   Matrix A, B, C, D, E, res;       /* test matrices */

   public MatrixTest(String testcase)
   {
      super(testcase);

      double a[][]=new double[][]{{9,6},{7,5}};
      A = new Matrix(2,2,a);
      double b[][]=new double[][]{{16,21},{3,12}};
      B = new Matrix(2,2,b);
      double d[][]=new double[][]{{2,2,3},{4,8,6},{7,8,9}};
      res=new Matrix();
   }

   public void testAdd()
   {
      double c[][]=new double[][]{{25,27},{10,17}};
      C = new Matrix(2,2,c);
      res=A.add(B);
      assertTrue(res!=null);
      assertTrue(C.equals(res));
   }

   public void testSetGet()
   {
      C=new Matrix(2,2);
      for (int i=0;i<2;i++)
         for (int j=0;j<2;j++)
            C.setElement(i,j,A.getElement(i,j));
      assertTrue(C.equals(A));
   }

   public void testMul()
   {
      double c[][]=new double[][]{{162,261},{127,207}};
      C = new Matrix(2,2,c);
      res=A.mul(B);
      assertTrue(res!=null);
      assertTrue(C.equals(res));
   }
```

Figure 9.12: Testing the matrix class with Junit.

```
    public void testTranspose()
    {
       res=A.transpose();
       res=res.transpose();
       assertTrue(res.equals(A));
    }

    public void testInverseCorrectness()
    {
       try{
          res=null;
          res=A.inverse();
          res=res.mul(A);
          double dd[][]=new double[][]{{1,0},{0,1}};
          Matrix DD=new Matrix(2,2,dd);
          assertTrue(res.equals(DD));
       }
       catch (Exception e)
       {assertTrue(false);
       }
    }
}
```

Figure 9.13: Testing the matrix class with Junit (contd.)

a project. We will first address the comparison issue. For this purposes we consider two approaches to testing—white box or structural testing separately. We will discuss these in detail in next chapter. For now, it is sufficient to say that black box testing is done without the knowledge of the internals of the programs while white box testing is driven by the internal structure of the programs.

By effectiveness, we mean the fault detecting capability. The effectiveness of a technique for testing a particular software will, in general, depend on the type of errors that exist in the software, as, in general, no one strategy does better than another strategy for all types of errors. Based on the nature of the techniques one can make some general observations about the effectiveness for different types of errors. One such comparison is given in Figure 9.14 [52].

As we can see, according to this comparison, different techniques have different strengths and weaknesses. For example, white box testing, as one would expect, is good for detecting logic errors, but not very good for detecting data handling errors. For data handling type errors, static analysis is quite good. Similarly, black box testing is good for input/output errors as it focuses on the external behavior, but it is not as good for detecting logic errors. As the figure shows, no one technique is good at detecting all types of errors, and hence no one technique can suffice for proper verification and validation. If high reliability is desired for the software, a combination of these techniques will have

	Technique				
Defect	Code Review	Static Analysis	Proof	White box Test	Black box Test
Computational	Med	Med	High	High	Med
Logic	Med	Med	High	High	Med
I/O	High	Med	Low	Med	High
Data handling	High	High	Med	Low	High
Interface	High	High	low	High	Med
Data Definition	Med	Med	Med	Low	Med
Database	High	Low	Low	Med	Med

Figure 9.14: Comparison of the different techniques.

to be used. From the table, one can see that if code reviews, white box testing, and black box testing are all used, then together they have a high capability of detecting errors in all the categories described earlier.

Another way of measuring effectiveness is to consider the "cost effectiveness" of different strategies, that is, the cost of detecting an error by using a particular strategy. And the cost includes all the effort required to plan, execute the verification approach, and evaluate the results. In cost effectiveness, static analysis fares the best, as without any human effort it can detect anomalies that have a high probability of containing errors. However, as many of these tools also have "false positives" which have to be evaluated before they can be identified as false positives, the effort required is not as small as it may look.

Code reviews can also be cost effective as they find faults directly, unlike in testing where only the failure is detected and the fault has to be found through debugging. Furthermore, no test case planning, test case generation, or test case execution is needed. However, reviews require considerable effort by a group of reviewers reviewing the code. Testing tends to be very cost effective for detecting the earlier defects, but as the remaining defects reduce, uncovering defects by testing becomes much harder. Formal verification is generally the most expensive as it is mostly human effort, and quite intense.

Let us now discuss how these techniques can be combined. It is clear from the comparison in Figure 9.14 and from the nature of white box and black box testing approaches, that the two basic approaches to testing are actually complementary. One looks at one program from the outside, the other from the inside. Hence, for effective testing of programs, both techniques should be applied. An approach to combine them is to start with selecting a set of test cases for performing the black box testing. These test cases will provide some coverage but may not provide the desired level of coverage.

The set of test cases is then augmented with additional test cases so that the desired coverage level is achieved.

Overall, first the available tools should be applied first, as they are probably the least expensive in terms of detecting defects. If unit testing and inspections both are to be done, then which one should be done first will probably depend on the situation. Generally, it is believed that inspections should be done first and then unit testing should be done. However, as inspections tend to be expensive and may be done only on critical code, it may be appropriate if inspections are done after some amount of unit testing has been done. Proofs of correctness are very labor intensive and are applied only for very critical programs.

9.5 Metrics

Traditionally, work on metrics has focused on the final product, namely the code. In a sense, all metrics for intermediate products of requirements and design are basically used to ensure that the final product has a high quality and the productivity of the project stays high. That is, the basic goal of metrics for intermediate products is to predict or get some idea about the metrics of the final product. For the code, the most commonly used metrics are size, complexity, and reliability. We will discuss reliability in the next chapter, as most reliability models use test data to assess reliability. Here we discuss a few size and complexity measures.

9.5.1 Size Measures

Size of a product is a simple measure, which can be easy to calculate. The main reason for interest in size measures is that size is the major factor that affects the cost of a project. Size in itself is of little use; it is the relationship of size with the cost and quality that makes size an important metric. It is also used to measure productivity during the project (e.g., KLOC per person-month). Final quality delivered by a process is also frequently normalized with respect to size (number of defects per KLOC). For these reasons, size is one of the most important and frequently used metrics.

The most common measure of size is delivered lines of source code, or the number of lines of code (LOC) finally delivered. The trouble with LOC is that the number of lines of code for a project depends heavily on the language used. For example, a program written in assembly language will be large compared to the same program written in a higher-level language, if LOC is used as a size measure. Even for the same language, the size can vary considerably depending on how lines are counted. Despite these deficiencies, LOC remains a handy and reasonable size measure that is used extensively. Currently, perhaps the most widely used counting method for determining the size is to count non-comment, non-blank lines only.

Halstead [79] has proposed metrics for length and volume of a program based on the number of operators and operands. In a program we define the following measurable quantities:

- n_1 is the number of distinct operators
- n_2 is the number of distinct operands
- $f_{1,j}$ is the number of occurrences of the jth most frequent operator
- $f_{2,j}$ is the number of occurrences of the jth most frequent operand

Then the vocabulary n of a program is defined as

$$n = n_1 + n_2.$$

With the measurable parameters listed earlier, two new parameters are defined:

$$N_1 = \sum f_{1,j}, N_2 = \sum f_{2,j}.$$

N_1 is the total occurrences of different operators in the program and N_2 is the total occurrences of different operands. The length of the program is defined as

$$N = N_1 + N_2.$$

From the length and the vocabulary, the volume V of the program is defined as

$$V = N log_2(n).$$

This definition of the volume of a program represents the minimum number of bits necessary to represent the program. $Log_2(n)$ is the number of bits needed to represent every element in the program uniquely, and N is the total occurrences of the different elements. Volume is used as a size metric for a program. Experiments have shown that the volume of a program is highly correlated with the size in LOC.

9.5.2 Complexity Metrics

The productivity, if measured only in terms of lines of code per unit time, can vary a lot depending on the complexity of the system to be developed. Clearly, a programmer will produce a lesser amount of code for highly complex system programs, as compared to a simple application program. Similarly, complexity has great impact on the cost of maintaining a program. To quantify complexity beyond the fuzzy notion of the ease with which a program can be constructed or comprehended, some metrics to measure the complexity of a program are needed.

Some metrics for complexity were discussed in Chapter 8. The same metrics that are applicable to detailed design can be applied to code. One such complexity measure discussed in the previous chapter is *cyclomatic complexity*, in which the complexity of a module is the number of independent cycles in the flow graph of the module. A number of metrics have been proposed for quantifying the complexity of a program [80], and studies have been done to correlate the complexity with maintenance effort. Here we discuss a few more complexity measures. Most of these have been proposed in the context of programs, but they can be applied or adapted for detailed design as well.

Size Measures

A complexity measure tries to capture the level of difficulty in understanding a module. In other words, it tries to quantify a cognitive aspect of a program. It is well known that, in general, the larger a module, the more difficult it is to comprehend. Hence, the size of a module can be taken as a simple measure of the complexity of the module. It can be seen that, on an average, as the size of the module increases, the number of decisions in it are likely to increase. This means that, on an average, as the size increases the cyclomatic complexity also increases. Though it is clearly possible that two programs of the same size have substantially different complexities, in general, size is quite strongly related to some of the complexity measures.

Halstead's Measure

Halstead also proposed a number of other measures based on his software science [79]. Some of these can be considered complexity measures. As given earlier, a number of variables are defined in software science. These are n_1 (number of unique operators), n_2 (number of unique operands), N_1 (total frequency of operators), and N_2 (total frequency of operands). As any program must have at least two operators—one for function call and one for end of statement—the ratio $n_1/2$ can be considered the relative level of difficulty due to the larger number of operators in the program. The ratio N_2/n_2 represents the average number of times an operand is used. In a program in which variables are changed more frequently, this ratio will be larger. As such programs are harder to understand, *ease of reading or writing* is defined as

$$D = \frac{n_1 * N_2}{2 * n_2}.$$

Halstead's complexity measure focused on the internal complexity of a module, as does McCabe's complexity measure. Thus the complexity of the module's connection with its environment is not given much importance. In Halstead's measure, a module's connection with its environment is reflected in terms of operands and operators. A call to another module is considered an operator, and all the parameters are considered operands of this operator.

Live Variables

In a computer program, a typical assignment statement uses and modifies only a few variables. However, in general the statements have a much larger context. That is, to construct or understand a statement, a programmer must keep track of a number of variables, other than those directly used in the statement. For a statement, such data items are called *live variables*. Intuitively, the more live variables for statements, the harder it will be to understand a program. Hence, the concept of live variables can be used as a metric for program complexity.

First let us define *live variables* more precisely. A variable is considered live from its first to its last reference within a module, including all statements between the first and last statement where the variable is referenced. Using this definition, the set of live variables for each statement can be computed easily by analysis of the module's code. The procedure of determining the live variables can easily be automated.

For a statement, the number of live variables represents the degree of difficulty of the statement. This notion can be extended to the entire module by defining the average number of live variables. The average number of live variables is the sum of the count of live variables (for all executable statements) divided by the number of executable statements. This is a complexity measure for the module.

Live variables are defined from the point of view of data usage. The logic of a module is not explicitly included. The logic is used only to determine the first and last statement of reference for a variable. Hence, this concept of complexity is quite different from cyclomatic complexity, which is based entirely on the logic and considers data as secondary.

Another data usage-oriented concept is *span*, the number of statements between two successive uses of a variable. If a variable is referenced at n different places in a module, then for that variable there are $(n - 1)$ spans. The average span size is the average number of executable statements between two successive references of a variable. A large span implies that the reader of the program has to remember a definition of a variable for a larger period of time (or for more statements). In other words, span can be considered a complexity measure; the larger the span, the more complex the module.

Knot Count

A method for quantifying complexity based on the locations of the control transfers of the program has been proposed in [149]. It was designed largely for FORTRAN programs, where explicit transfer of control is shown by the use of goto statements. A programmer, to understand a given program, typically draws arrows from the point of control transfer to its destination, helping to create a mental picture of the program and the control transfers in it. According to this metric, the more intertwined these arrows become, the more complex the program. This notion is captured in the concept of a "knot."

A *knot* is essentially the intersection of two such control transfer arrows. If each statement in the program is written on a separate line, this notion can be formalized as follows. A jump from line a to line b is represented by the pair (a, b). Two jumps (a, b) and (p, q) give rise to a knot if either min (a, b) < min (p, q) < max (a, b) and max (p, q) > max (a, b); or min (a, b) < max (p, qa) < max (a, b) and min (p, q) < min (a, b).

Problems can arise while determining the knot count of programs using structured constructs. One method is to convert such a program into one that explicitly shows control transfers and then compute the knot count. The basic scheme can be generalized to flow graphs, though with flow graphs only bounds can be obtained.

Topological Complexity

A complexity measure that is sensitive to the nesting of structures has been proposed in [31]. Like cyclomatic complexity, it is based on the flow graph of a module or program. The complexity of a program is considered its maximal intersect number *min*.

To compute the maximal intersect, a flow graph is converted into a strongly connected graph (by drawing an arrow from the terminal node to the initial node). A strongly connected graph divides the graph into a finite number of regions. The number of regions is (edges - nodes + 2). If we draw a line that enters each region exactly once, then the number of times this line intersects the arcs in the graph is the maximal intersect *min*, which is taken to be the complexity of the program.

9.6 Summary

The goal of the coding activity is to develop correct programs that are also clear and simple. As reading programs is a much more common activity than writing programs, the goal of the coding activity is to produce simple programs that are easy to understand and modify and that are free from errors. Ease of understanding and freedome from defects are the key properties of high quality code. The focus of this chapter is to discuss approaches that can be used for developing high quality code.

We discussed some principles whose application can help improve code quality. These include structured programming and information hiding. In structured programs, the program is a sequence of single-entry, single-exit statements, and the control flow during execution is linearized. This makes the dynamic structure of a program similar to the static structure, and hence easy to understand and verify. In information hiding, the data structures are hidden behind access functions, thereby raising the level of abstraction and hiding complexity. We also described some common coding errors to make the programmer aware about them, and some programming practices that can help reduce errors. We briefly discussed coding standards that help improve readability as well as reduce errors.

There are different ways a programmer can proceed with developing code. We discussed a few processes that can be followed by a programmer. One is the incremental process in which the programmer writes code in small increments and tests and debugs each increment before writing more code. Test driven development is a programming approach in which test cases are written first and then code is written to pass these test cases. When the code succeeds, the programmer writes another small set of test cases and then the code to implement it. Test driven development is also an incremental programming approach. Pair programming is another approach, in which coding is done by a pair of programmers. Both programmers together discuss the strategy, data structures, and algorithms to be used. When one programmer does the actual coding, the other reviews it as it is being typed. Regardless of the approach the programmer follows, source code control is an important part of the coding process. We briefly discussed how source code control is used by programmers.

Code evolves and changes over time as systems evolve. Often due to these changes the design of the software becomes too complex making changing harder. Refactoring is an approach in which during the coding activity, effort is spent in refactoring the existing code to improve its design so that changes become easier to make. During refactoring no new functionality is added—only improvement is done so that the design of the code improves by reduction of coupling, increase in cohesion, and better use of hierarchies. We have discussed the refactoring process and various techniques for doing refactoring, and have given a detailed example.

The code written by a programmer should be verified before it is incorporated in the overall system. The most commonly used techniques for verification are static analysis, code inspections, and unit testing. In static analysis, the source code is examined for anomalies and situations that can represent bugs through suitable tools. It is a very efficient way of detecting errors and static analyzers can detect a variety of defects. However, warnings given by a static analyzers also have "false positives," requiring further analysis by the programmers to identify actual defects. Code inspections, using a standard inspection process, is a very effective approach for finding errors. It can, however, be expensive, and hence is sometimes replaced with code reading or one-person review. Unit testing the code is a very popular and most often used practice by programmers. In this drivers and stubs are written to test the program the programmer had developed. Unit testing can benefit from tools, and we have briefly discussed how unit testing can be done with the Junit tool. Formal verification is another approach that is sometimes used for very critical portions of the system. We have discussed one approach for formal verification also.

A number of metrics exist for quantifying different qualities of the code. The most commonly used are size metrics, because they are used to assess the productivity of people and are often used in cost estimation. The most common size measure is lines

of code (LOC), which is also used in most cost models. There are also other measures for size. The goal of complexity metrics is to quantify the complexity of software. Complexity is an important factor affecting the productivity of projects and is a factor in cost estimation. A number of different metrics exist. Perhaps the most common is the cyclomatic complexity, which is based on the internal logic of the program and defines complexity as the number of independent cycles in the flow graph of the program.

Exercises

1. What is structured programming and how does it help improve code quality?
2. If you have all the tools available, how will you do the verification of the programs you write?
3. For memory and resource related errors (memory leaks, null dereferencing, etc.) compare the effectiveness and efficiency of the different verification techniques.
4. Buffer overflow is a common error which is also a main security flaw. What are some of the coding practices that will help minimize this error? What other types of errors these practices will impact?
5. Draw a flow diagram describing your own personal process. Critically evaluate it and suggest modifications that will help improve the quality of the code you write.
6. Work on some programs alone. Then, along with a friend, develop some other programs using pair programming. Compare the productivity and code quality achieved in the two approaches.
7. In your next project, develop a few classes using your standard approach. Then use Junit and develop a few classes using an incremental approach. Record effort and defect data and then compare the average development time, productivity, and defects found.
8. Do a similar experiment with TDD. How does it compare with your regular development process?
9. What are the major concepts that help make a program more readable?
10. Consider the following program to determine the product of two integers x and y:

```
if (x = 0) or (y = 0) then
        p := 0
else begin
```