CODING & DOCUMENTATION PHASE

The goal of the coding or programming activity is to implement the design in the best possible manner. The coding activity affects both testing and maintenance profoundly. The time spent in coding is a small percentage of the total software cost, while testing and maintenance consume the major percentage. Thus, it should be clear that the goal during coding should not be to reduce the implementation cost, but the goal should be to reduce the cost of later phases, even if it means that the cost of this phase has to increase. In other words, the goal during this phase is not to simplify the job of the programmer. Rather, the goal should be to simplify the job of the tester and the maintainer. During coding, it should be kept in mind that the programs should not be constructed so that they are easy to write, but so that they are easy to read and understand. A program is read a lot more often and by a lot more people during the later phases.

Structured Programming
Structured programming is often regarded as "goto-less" programming. Although extensive use of gotos is certainly not desirable, structured programs can be written with the use of gotos. A program has a static structure as well as a dynamic structure. The static structure is the structure of the text of the program, which is usually just a linear organization of statements of the program. The dynamic structure of the program is the sequence of statements executed during the execution of the program. The closer the correspondence between execution and text structure, the easier the program is to understand, and the more different the structure during execution, the harder it will be to argue about the behavior from the program text. The goal of structured programming is to ensure that the static structure and the dynamic structures are the same. That is, the objective of structured programming is to write programs so that the sequence of statements executed during the execution of a program is the same as the sequence of statements in the text of that program. As the statements in a program text are linearly organized, the objective of structured programming becomes developing programs whose control flow during execution is linearized and follows the linear organization of the program text. No meaningful program can be written as a sequence of simple statements without any branching or repetition (which also involves branching). So, how is the objective of linearizing the control flow to be achieved? By making use of structured constructs. In structured programming, a statement is not a simple assignment statement, it is a structured statement. The key property of a structured statement is that it has a single-entry and a single-exit That is, during execution, the execution of the (structured) statement starts from one defined point and the execution terminates at one defined point. With single-entry and single-exit statements, we can view a program as a sequence of (structured) statements. And if all statements are structured statements, then during execution, the sequence of execution of these statements will be the same as the sequence in the program text. Hence, by using single-entry and single-exit statements, the correspondence between the static and dynamic structures can be obtained. The most commonly used single-entry and single-exit statements are:

Selection: if B then SI else S2
if B then SI
Iteration: While B do S
repeat S until B
Sequencing: SI; S2; S3...

It can be shown that these three basic constructs are sufficient to program any conceivable algorithm. Modern languages have other such constructs that help linearize the control flow of a program, which, generally speaking, makes it easier to understand a program. Hence, programs should be written so that, as far as possible, single-entry, single-exit control constructs are used. The main reason structured programming was promulgated is formal verification of programs. During verification, a program is considered a sequence of executable statements, and verification proceeds step by step, considering one statement in the statement list (the program) at a time. Implied in these verification methods is the assumption that during execution, the statements will be executed in the sequence in which they are organized in the program text. If this assumption is satisfied, the task of verification becomes easier. Hence, even from the point of view of verification, it is important that the sequence of execution of statements is the same as the sequence of statements in the text. Any piece of code with a single-entry and single-exit cannot be considered a structured construct. If that is the case, one could always define appropriate units in any program to make it appear as a sequence of these units (in the worst case, the whole program could be defined to be a unit). The basic objective of using structured constructs is to linearize the control flow so that the execution behavior is easier to understand and argue about. In linearized control flow, if we understand the behavior of each of the basic constructs properly, the behavior of the program can be considered a composition of the behaviors of the different statements. It should be kept in mind that structured programming is not an end in itself. Our basic objective is that the program be easy to understand. And structured programming is a safe approach for achieving this objective. Any unstructured construct (such as: break statement, continue statement) should be used only if the structured alternative is harder to understand.

Modular Programming

Modular programming is subdividing the program into separate subprograms such as functions and subroutines. For example, if the program needs initial and boundary conditions, use subroutines to set them. Then if someone else wants to compute a different solution using the program, only these subroutines need to be changed. This is a lot easier than having to read through a program line by line, trying to figure out what each line is supposed to do and whether it needs to be changed. And in ten years after development, developers will probably no longer remember how the program worked. Subprograms make the actual program shorter, hence easier to read and understand. Further, the arguments show exactly what information a subprogram is using. That makes it easier to figure out whether it needs to be changed when you are modifying your program. Forgetting to change all occurrences of a variable is a very common source of errors. Subprograms make it simpler to figure out how the program operates. If the boundary conditions are implemented using a subroutine, your program can be searched for this subroutine to find all places where the boundary conditions are used. This might include some unexpected places, such as in the output, or in performing a numerical check on the overall accuracy of the program. Subprograms reduce the likelihood of bugs. Since subprograms can use local variables, there is less change that the code in the subroutine interferes with that of the program itself, or with that in other subprograms. The smaller size of the individual modules also makes it easier to understand the global effects of changing a variable. Function of generating output can be performed using a subprogram. One tends to become more careless when programming a `noncritical' program part such as output. Also, output often requires additional variables not of interest to the rest of the program. The code for output tends to be lengthy; hence the program will become shorter and easier to read if output is moved to a subroutine. Use the correct type of subprogram. For example, if your program uses an exact solution, cast it into the form of a function subprogram, not a subroutine. Clarity and ease of use is improved by using modular programming. Modular programming is a solution to the problem of very large programs that are difficult to debug and maintain. By segmenting the program into modules that perform clearly defined functions, one can determine the source of program errors more easily. Object-orientated programming languages, such as SmallTalk and HyperTalk, incorporate modular programming principles.
Regarding implementing concept of modular programming, message passing has more recently, gained ground over the earlier and more conventional "Call" interfaces method, thus, becoming the more dominant linkage between separate modules. This has solved the "versioning problem" (sometimes experienced when using interfaces for communication between the modules).

Coupling-Cohesion and Refactoring

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, 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. 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 refactoring can significantly transform the program structure. With refactoring, code becomes continuously improved. 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. 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 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. Refactoring is not a technique for bug fixing or for improving code that is in very bad shape.

Object Oriented Programming (OOP)

If you use classes and objects in your programs, but neither inheritance nor polymorphism, many authors refer to this as "object-based" programming, but insist that it not be called "object-oriented" programming unless you employ inheritance and polymorphism as well. The 3 key concepts in OOP are: Encapsulation, Inheritance and Polymorphism. Encapsulation in this context means "putting together the things that should be together, in particular attributes and operations (data and methods). The fundamental underlying notion is that of "abstraction", and in particular the Abstract Data Type (ADT). An ADT is usually implemented by something called a "class". Data (attributes) and operations (behaviors, and also called functions, or methods) are combined in the class definition. Both are also referred to as "members" of the class. A program consists of a collection of "objects" of various classes that make things happen by

communicating with each other by "sending messages"; i.e., one object can send a message to another object by asking that object to perform one of its methods (Java syntax: object.method(parameter_list))
The Principle of Information Hiding insists on a "contract" between the implementor and the user. This establishes a "division of responsibility" in which the implementor should be told only what is necessary for implementing the class, and the client should be told only what is necessary to use the class. This may be enforced, or at least encouraged by "access control" (public, private, protected) of class members. Inheritance is a hierarchical relationship in which the members of one class are all passed down to any class that descends from (extends) that class, directly or indirectly. This is just one of a number of different relationships that may exist between different classes. Thus, inheritance is a mechanism for defining a new class based on the definition of a pre-existing class, in such a way that all the members of the "old" class (superclass, or parent class, or base class) are present in the "new" class (subclass, or child class, or derived class), and an object of the new class may be substituted anywhere for an object of the old class. This is the Principle of Substitutability. The following examples represent the "is-a" relationship:

- A CommissionEmployee is-an Employee.
- A SalariedEmployee is-an Employee.
- A Tiger is-a Mammal.
- A Whale is-a Mammal.

An inherited class may simply use the members that are already there, may add new members, or may "override" members that pre-existed in the parent or ancestor class (by giving those "overridden" members new definitions). An inheritance hierarchy is a tree-like mapping of the relationships that form between classes as the result of inheritance. (C++ allows multiple inheritance, as in "a SingingWaiter is-a Singer and a SingingWaiter is-a Waiter", but this is not permitted in Java.) Inheritance hierarchies should develop naturally as you program, and not be "forced" in any way. There are different "kinds" of inheritance:

Specialization: In this form of inheritance the derived class adds more functionality and/or overrides some of the existing functionality. It makes good use of polymorphism, and is probably the most frequently used form of inheritance.

Extension: This is a particular form of specialization in which more functionality is added but none of the existing functionality is overridden.

Realization of a Specification: In this form an "abstract class" or "interface" (both of which contain unimplemented methods) is used as the base to realize a concept merely "specified" in that base.

Combination: This occurs when we want to use more than one base class The alternative is to "inherit from", or "extend", a single base class, and simultaneously "implement" one or more "interfaces".

Other ways to use inheritance, should be avoided, or at least considered very carefully before use. Polymorphism literally means "many forms". Here is an everyday example: We use the word "open" in many ways (many forms), since we open doors, open windows, open envelopes, open packages, open bank accounts, open our minds, open our hearts, open our eyes, and open our arms. In OOP, polymorphism refers to the fact that a single name (like open, above) can be used to represent or select different code at different times, depending on the situation, and according to some automatic mechanism. So, for example, the method call object.Area() can mean different things at different times in the same program. There are different kinds of polymorphism:

Pure polymorphism: This is also called inclusion polymorphism), and means that a single function is applied to a variety of types (Example: the valueOf method in class String)

Overriding: In this case a method further up a class hierarchy is redefined in a subclass, giving either total replacement of the redefined method, or just a refinement of that method.

Deferred methods: These are methods that are specified but not implemented in an abstract class or an interface, and in one sense this may be considered a generalization of overriding.

Overloading: This is also called ad hoc polymorphism), and in this case a number of different functions (code bodies) all have the same name, but are distinguished by having different parameter lists (parametric overloading, often aided in C++ by the use of default parameters, which are not available in Java) Note, however, that overloading does not necessarily imply similar actions by the code. (Example: In a card game program, draw might mean to draw a card on the screen in one class, but to choose a card from the deck in another class.)

Some of the *b*enefits of OOP are:

- *Natural*: OOP uses the terminology of the problem, not the terminology of a computer.
- *Reliable*: Modular construction puts knowledge and responsibility where they belong, and allows each component to be tested and validated independently.
- *Reusable*: Well designed class objects can form reusable components (but of course this is not guaranteed).
- *Maintainable*: OOP permits software components to change or even be replaced completely in a way that should be transparent to the rest of the system.
- *Extendable*: OOP permits easy addition of new functionality, as the user's needs change over time.
- *Timely*: OOP permits quicker development times by allowing parallel development of independent classes by independent developers.

While the pitfalls are:

- *Thinking only of the programming language:* That is, not paying enough attention to everything else that's involved, and then blaming "the technology" when things go wrong.
- *Thinking of OOP as a cure-all*: Using OPP does not guarantee success. There is never any substitute for good judgment. Planning, designing and careful coding are always necessary, whatever your approach and however sophisticated your tools may be.
- *Fearing to re-use code*: That is, always wanting to start from scratch.
- *Selfish programming*: That is, not wanting to share the code you create.

Information Hiding

A software solution to a problem always contains data structures that are meant to represent information in the problem domain. That is, when software is developed to solve a problem, the software uses some data structures to capture the information in the problem domain. In general, only certain operations are performed on some information. That is, a piece of information in the problem domain is used only in a limited number of ways in the problem domain. For example, a ledger in an accountant's office has some defined uses: debit, credit, check the current balance, etc. An operation where all debits are multiplied together and then divided by the sum of all credits is typically not performed. So, any information in the problem domain typically has a small number of defined operations performed on it. When the information is represented as data structures, the same principle should be applied, and only some defined operations should be performed on the data structures. This, essentially, is the principle of information hiding. The information captured in the data structures should be hidden from the rest of the system, and only the access functions on the data structures that represent the operations performed on the information should be visible. In other words, when the information is captured in data structures and then on the data structures that represent some information, for each operation on the information an access function should be provided. And as the rest of the system in the problem domain only performs these defined operations on the information, the rest of the modules in the software should only use these access functions to access and manipulate the data structures.

Information hiding can reduce the coupling between modules and make the system more maintainable. Information hiding is also an effective tool for managing the complexity of developing software—by using information hiding we have separated the concern of managing the data from the concern of using the data to produce some desired results. Many of the older languages, like Pascal, C, and FORTRAN, do not provide mechanisms to support data abstraction. With such languages, information hiding can be supported only by a disciplined use of the language. That is, the access restrictions will have to be imposed by the programmers; the language does not provide them. Most modern OO languages provide linguistic mechanisms to implement information hiding.

Reuse

Code reuse, also called software reuse, is the use of existing software, or software knowledge, to build new software. Therefore, software reuse is the process of creating software systems from existing software rather than building them from scratch. Code scavenging, using source code generators, or reusing knowledge can contribute to increased productivity in software development. Advances in software engineering have contributed to increased productivity. Examples include high-level programming languages, object-oriented technology, prototyping-oriented software development, and computer-aided software engineering. Studies on reuse have shown that 40% to 60% of code is reusable from one application to another, 60% of design and code are reusable in business applications, 75% of program functions are common to more than one program, and only 15% of the code found in most systems is unique and novel to a specific application. Maximizing the reuse of tested (if not verified or certified) source code and minimizing the need to develop new code alone can bring improvements in cost, time and quality, and reusing source code is only the (low-level) beginning of reuse. Practicing reuse systematically requires additional effort, e.g., managerial and organizational changes. Concerning motivation and driving factors, reuse can be:

- Opportunistic - While getting ready to begin a project, the team realizes that there are existing components that they can reuse.
- Planned - A team strategically designs components so that they'll be reusable in future projects.

Reuse can be categorized further:

- Internal reuse - A team reuses its own components. This may be a business decision, since the team may want to control a component critical to the project.
- External reuse - A team may choose to license a third-party component. Licensing a third-party component typically costs the team 1 to 20 percent of what it would cost to develop internally. The team must also consider the time it takes to find, learn and integrate the component.

Concerning form or structure of reuse, code can be:

- Referenced - The client code contains a reference to reused code, and thus they have distinct life cycles and can have distinct versions.
- Forked - The client code contains a local or private copy of the reused code, and thus they share a single life cycle and a single version.

Fork-reuse is often discouraged because it's a form of code duplication, which requires that every bug is corrected in each copy, and enhancements made to reused code need to be manually merged in every copy or they become out-of-date. However, fork-reuse can have benefits such as isolation, flexibility to change the reused code, easier packaging, deployment and version management.

Documentation

Software systems contain all relevant 'information' in order to be executable on a machine. Human readers need additional information which has to be provided in the documentation of a software system. Documentation has to be produced during the software process for various categories of readers. All statements made subsequently about documentation apply to software components and/or software systems. The documentation of software systems composed of software components should be a collection of the components' documentation plus additional information. This means that the documentation of a component has to fulfill requirements similar to those on the component itself. For example, the documentation should be self contained, adaptable and extensible (in case the component is being adapted and/or extended). Information has to be provided for end users, management, developers and maintenance personnel. Different reader groups have different information needs which are addressed in different kinds of information. End users need information that enables them to efficiently and effectively use and administer the system (user documentation). Management needs help for planning, budgeting and scheduling current and future (similar) software processes (process documentation). Developers need information about the overall system structure, about components and their interaction (system documentation).

User documentation: Good user documentation is important for the commercial success of a software product. Users need different kinds of information and there are different kinds of users, e.g., novice and experienced users. Users must be able to use a software system with the information provided in the user documentation. Additional assistance and/or further information should not be necessary. A component may or may not be (directly) used by end users; thus user documentation of components is optional. However, components that are not full-fledged applications but do interact with users may have user documentation. The user documentation of an application being composed of such components may be composed of the documentation of these components. Five parts for user documentation are required generally. Depending on the size and kind of a software system, some of these may be optional or be combined with other parts. The five parts are:

1) *Functional description*: outline of system requirements and provided services (for system evaluation)
2) *Installation manual*: detailed information on how to install the system in a particular environment (for system administrators)
3) *Introductory manual*: informal introduction to the system and description of standard features (for novice users)
4) *Reference manual*: complete description of all features and error messages (for experienced users)
5) *System administrator manual*: more general information for system administration (for system administrators)

System documentation has to capture all information about the development of a software component/system. It should contain sufficient information such that a new member of the development or maintenance team can make modifications and extensions without further information and/or assistance. System documentation includes the following information:

1) *Requirements*: contract between component user (end user and/or reuser) and component developer
2) *Overall design and structure*: subcomponents and their interrelations
3) *Implementation details*: e.g., algorithmic details
4) *Test plans* and reports: for integration tests and acceptance tests
5) *Used files*: in case component/system uses external files
6) *Source code listings*: too often the only accurate and complete description of a component or system

System documentation of components composed of smaller-grained components typically contains the documentation of the components it is composed of. It is important for documentation composition that a component's documentation be self-contained, modifiable and extensible. Due to time pressure, system documentation is often neglected. Thus information is often incomplete and inconsistent. Literate programming provides a means for improving documentation completeness and consistency.

Process documentation: In contrast to user and system documentation, which describe a component at a certain point of time, process documentation describes the dynamic process of its creation. The reason for documenting the process is to support effective management and project control. Whereas user and system documentation have to be kept up-to-date, much of the process documentation becomes outdated. Process documentation is a collection of information that depicts the whole development process. Its documents include the following:

1) *Project plan*: individual phases with estimates and schedules (for prediction and control)
2) *Organization plan*: allocation and supervision of personnel
3) *Resource plan*: allocation and supervision of resources other than personnel (e.g., machine time, travelling expenditures)
4) *Project standards*: e.g., design methodology, test strategy, documentation conventions

5) *Working papers*: technical communication documents that record ideas, strategies, and identified problems (contain information about the rationale of design decisions)
6) *Log book*: discussions and communication between project members (design decisions)
7) *Reading aids*: e.g., index of documents, word index, table of contents etc.

Process documentation helps in controlling the current project, i.e., controlling project progress (project plans and estimates), controlling quality (standards, check points), and determining production costs (personnel cost, machine time). Another reason for the creation of this kind of documentation is that its information can be used when creating similar components or systems.

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.

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, MAXJTERATIONS)
- 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., OutOfBound-Exception.)

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.

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. Techniques that are widely used in practice are inspections (including code reading), unit testing, and program checking.

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:

- 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?

Inspections are very effective for detecting defects and are widely used in many commercial organizations.

Code Reading

Code inspections tend 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 structured code reading technique. 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*.

Code Walkthroughs

The purpose of the code walkthrough is to ensure that the requirements are met, coding is sound, and all associated documents completed. Design is presented at the code walkthrough in order for the Reviewer to understand the context of the software change. Depending upon the scope of the software change, the design does not need to be written, but may be presented orally. Code is reviewed for the following:

- Clarity and Readability - as in accordance to the coding standards.
- Requirements Met - code does what it is supposed to do.
- Performance - code is coded to prevent performance problems
- Algorithms and Error Situations - algorithm used is sound, error conditions handled, all reasonable conditions are handled.

In appropriate cases, the old and new code may be presented in order to show the differences, if it helps explain the changes. User documentation, if appropriate, is examined for:

- Existence, for the functionality implemented.
- Clarity and Readability
- Completeness - e.g., User, Training, Reference, Table of Contents

Release note documentation, if appropriate, is examined for:

- Existence, for the functionality implemented.
- Clarity and Readability

Written Test Plans are reviewed, if appropriate, for:

- Existence, for the functionality implemented.
- Completeness - for a written test plan to cover the particular functionality or correction. All reasonable cases are handled.

Automated Test Plans are reviewed, if appropriate, for:

- Existence, for the functionality implemented.
- Completeness - for the various scenarios / use cases to exercise the software. All reasonable cases are handled.

If the task does not warrant a written or automated test plan, then the developer is expected to explain how the code has been tested. The reviewer should check that the code was:

- Exercised, for the functionality implemented.
- Completeness - for the various scenarios / use cases to exercise the software. All reasonable cases are handled.

At least two people are required for the code walkthrough:

- Developer - who wrote the code
- Reviewer - who reviews the code

The level of the code walkthrough is determined by the extent of the code changes to be reviewed. Complexity of the changes, extent of the changes, and criticality of the changes all play into the decision of place/location of the code walkthrough. The Developer makes a recommendation for the appropriate level of the code walkthrough. The Project Lead may override that recommendation. The various level/types of the code walkthrough are:

- Waived: Code walkthrough is waived due to a very simple algorithm change (or configuration file), and not in a critical point in the code.
- Informal: Code is presented to the reviewer who simply reviews it "on-the-spot". This is reserved for simple, confined, and non-critical changes.
- Formal: Code is presented to the reviewer and the reviewer examines the code and algorithms for proper style, format, and technique. This is reserved for complex code changes of a critical nature and those changes that may impact multiple subsystems.

In addition based on the complexity, extent, and criticality of the changes, as well as the reviewer's knowledge of the design and implementation, the code walkthrough may be conducted independently by the reviewer, rather than have the developer do a presentation or be present. If the reviewer has questions, then the reviewer is responsible for seeking out the developer to answer the questions.
There can be three outcomes to the code walkthrough:

- Successful: all required checks and quality are present. Software may be released into the approved build.
- Corrective Action Needed, No Further Review Needed: a list of items to be corrected are presented. Once these are performed, the code walkthrough is complete.
- Corrective Action Needed, Review Needed: a list of items to be corrected are presented. One these are performed, another code walkthrough is warranted.

If the code walkthrough was successful, then the software change may be released into the approved build.

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. A unit may be a function, a small collection of 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. 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.

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. 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. 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. The most widely used program checking technique is static analysis, which is becoming increasingly popular with more tools becoming available.
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 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 are 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. 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 on data flow anomalies is based on checkers described in, 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 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.
The 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, 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, 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-condition).

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.