Coding and Documentation

Software Engineering

Slides compiled by Sanghamitra De

Purpose of Coding

- Implement the design in the best possible manner
- To simplify the job of the tester and the maintainer.
- To write code which are easy to read and understand.

Structured Programming

- Often regarded as "goto-less" programming.
- Goal of structured programming is to ensure that the **static structure** and the **dynamic structures** of the program are the same.
- This is done by making use of *structured constructs*.
- By using single-entry and single-exit statements, correspondence between the static and dynamic structures can be obtained.
- As far as possible, single-entry, single-exit control constructs are used

Structured Programming

- Some 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...

Structured Programming

- From the point of view of formal verification of programs, it is important that the sequence of execution of statements is the same as the sequence of statements in the text.
- Basic objective of using structured constructs is to linearize the control flow so that the execution behavior is easier to understand and argue about.
- 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.
- Subprograms make the actual program shorter, hence easier to read and understand.
- Subprograms reduce the likelihood of bugs.
- Modular programming is a solution to the problem of very large programs that are difficult to debug and maintain.
- Regarding implementing concept of modular programming, message passing has more recently, gained ground over the earlier and more conventional "Call" interfaces method.

Coupling-Cohesion and Refactoring

- Refactoring is the technique to improve existing code and prevent 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.
- Refactoring generally results in one or more of the following:
  - ✓ Reduced coupling
  - ✓ Increased cohesion
  - ✓ Better adherence to open-closed principle (for OO systems)

Coupling-Cohesion and Refactoring

- The main risk of refactoring is that existing working code may "break" due to the changes being made.
- The other reason is that it may be viewed as an additional and unnecessary cost.
- To mitigate this risk, the two golden rules are:
  - ✓Refactor in small steps
  - ✓Have test scripts available to test existing functionality

Coupling-Cohesion and Refactoring

- By doing refactoring in a series of small steps, and testing after each step, mistakes in refactoring can be easily identified and rectified.
- With refactoring, the quality of the design improves, making it easier to make changes to the code as well as find bugs.

Object Oriented Programming (OOP)

- 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.
- 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).

Object Oriented Programming (OOP)

- A program consists of a collection of "objects" of various classes that make things happen by communicating with each other by "sending messages“.
- 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 is enforced 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.

Object Oriented Programming (OOP)

- So, 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**.

Object Oriented Programming (OOP)

- 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".

Object Oriented Programming (OOP)

- Polymorphism literally means "many forms".
- In OOP, polymorphism refers to the fact that a single name 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.

Object Oriented Programming (OOP)

- 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. Note, however, that overloading does not necessarily imply similar actions by the code.

Object Oriented Programming (OOP)

- Benefits of OOP are:
  - ✓ *Natural*
  - ✓ *Reliable*
  - ✓ *Reusable*
  - ✓ *Maintainable*
  - ✓ *Extendable*
  - ✓ *Timely*

Object Oriented Programming (OOP)

- Pitfalls of OOP are:
  - ✓*Thinking only of the programming language*
  - ✓*Thinking of OOP as a cure-all*
  - ✓*Fearing to re-use code*
  - ✓*Selfish programming*

Information Hiding

- When the information is represented as data structures, 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.

Information Hiding

- 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 separate the concern of managing the data from the concern of using the data to produce some desired results.

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.

Reuse

- 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

Reuse

- 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.

Reuse

- 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.

Documentation

- Documentation has to be produced during the software process for various categories of readers.
- The documentation of software systems composed of software components should be a collection of the components' documentation plus additional information.
- 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.

Documentation

- **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).

Documentation

- User documentation
- System documentation
- Process documentation

User documentation

- 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.

User documentation

- Five parts for user documentation are required generally.
  - ✓ *Functional description*: outline of system requirements and provided services (for system evaluation)
  - ✓ *Installation manual*: detailed information on how to install the system in a particular environment (for system administrators)
  - ✓ *Introductory manual*: informal introduction to the system and description of standard features (for novice users)
  - ✓ *Reference manual*: complete description of all features and error messages (for experienced users)
  - ✓ *System administrator manual*: more general information for system administration (for system administrators)

System documentation

- 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 of components composed of smaller-grained components typically contains the documentation of the components it is composed of.

System documentation

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

Process documentation

- Process documentation describes the dynamic process of creation of a component.
- The reason for documenting the process is to support effective management and project control.
- User and system documentation have to be kept up-to-date, but much of the process documentation becomes outdated.
- Process documentation is a collection of information that depicts the whole development process.
- 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).
- This kind of documentation can be used when creating similar components or systems.

Process documentation

- Process documents include the following:
  - ✓ *Project plan*: individual phases with estimates and schedules (for prediction and control)
  - ✓ *Organization plan*: allocation and supervision of personnel
  - ✓ *Resource plan*: allocation and supervision of resources other than personnel (e.g., machine time, travelling expenditures)
  - ✓ *Project standards*: e.g., design methodology, test strategy, documentation conventions
  - ✓ *Working papers*: technical communication documents that record ideas, strategies, and identified problems (contain information about the rationale of design decisions)
  - ✓ *Log book*: discussions and communication between project members (design decisions)
  - ✓ *Reading aids*: e.g., index of documents, word index, table of contents etc.

Coding Standards

- It is of prime importance to write code in a manner so 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.
- Coding standards provide guidelines for programmers regarding naming, file organization, statements and declarations, and layout and comments.

Naming Conventions

- 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.)

Conventions for Files

- 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.

Conventions for Statements

- 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.
- 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.
- 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.

Commenting and Layout

- Some guidelines on commenting:
  - ✓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.

Commenting and Layout

- 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.

Verification

- Once a programmer has written the code for a module, it has to be verified before it is used by others.
- Though testing is the most common method of verification, there are other effective techniques also.
- Code verification methods are:
  - ✓ Code Inspections
  - ✓ Code Reading
  - ✓ Code Walkthroughs
  - ✓ Unit Testing
  - ✓ Static Analysis

Code Inspections

- Code inspections are usually held after code has been successfully compiled and other forms of static tools have been applied.
- Aim is to save human time and effort, which would otherwise be spent detecting errors that a compiler or static analyzer can detect.
- In addition to defects, there are quality issues which code inspections usually look for, like efficiency, compliance to coding standards, etc.
- 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.

Code Inspections

- Typical items for a checklist of code review:
  - ✓ 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?

Code Reading

- 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.
- 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 sometimes called *desk review*.

Code Walkthroughs

- 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.
- 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.

Code Walkthroughs

- 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

Code Walkthroughs

- 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.

Code Walkthroughs

- 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

Code Walkthroughs

- The various level/types of the code walkthrough are:
  - ✓ Waived
  - ✓ Informal
  - ✓ Formal
- There can be three outcomes to the code walkthrough:
  - ✓ Successful
  - ✓ Corrective Action Needed, No Further Review Needed
  - ✓ Corrective Action Needed, Review Needed

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.
- 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.

Unit Testing

- 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.

Unit Testing

- The test data will include some data representing the normal case special cases which might result in special or exceptional result.
- 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.
- 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

- Checking the programs through the aid of analysis tools is called program checking.
- Three forms of checking:
  - ✓model checking
  - ✓dynamic analysis
  - ✓static analysis.

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 specified and a model checker checks whether the model satisfies the stated properties.

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.

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 Analysis

- Approaches:
  - ✓To detect patterns in code that are "unusual" or "undesirable" and which are likely to represent defects.
  - ✓Directly look for defects in the code, that is, look for those conditions that can cause programs to fail when executing.

Static Analysis

- Limitations of a static analyzer:
  - ✓ **Soundness** captures the occurrence of false positives in the errors the static analyzer identifies.
  - ✓ **Completeness** characterizes how many of the existing errors are missed by the static analyzer.

Static Analysis

- Forms of Static Analysis:
  - ✓ Looking for unusual patterns in code. 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.
- Redundancies identified by the checkers
  - ✓ Idempotent operations
  - ✓ Assignments that were never read
  - ✓ Dead code
  - ✓ Conditional branches that were never taken

Static Analysis

- Redundancies:
  - ✓ *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.

Static Analysis

- Second approach for static analysis is to directly look for errors in the programs - bugs that can cause failures when the programs execute.
- Tools are there which directly look for errors - the level of false positives generated by this tool tends to be low.
- Considered tool: PREfix

Static Analysis

- 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

Static Analysis

- 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.

Static Analysis

- 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).

Static Analysis

- Advantages of static analysis:
  - ✓ As static analysis is performed with the help of software tools, it is a ***cost-effective way of discovering errors***.
  - ✓ It detects the errors directly and not just the presence of errors, as is the case with testing. So, ***little debugging is needed*** after the presence of error is detected.
  - ✓ It is also a ***scalable technique of detecting errors*** in the code.

End …