Software quality assurance

Lecture 8

Instructor: Mr. Natash Ali Mian

Department of CS and IT
The University of Lahore

Switch off mobile phones during lectures, or put them into silent mode

Use Dustbins to throw waste materials

TERM PAPER

Finalize Group Members	26-Feb-2013
Finalize Topic	12-Mar-2013
Search Papers and Sort Selected (TODAY)	20-Mar-2013
Go Through the Abstract and Introduction of Selected Papers	27-Mar-2013
Submit a Summary and Comments on related papers	**09-Apr-2013**
Present Your Work till Today	**09-Apr-2013**
Submit Initial Draft	30-Apr-2013
Final Paper Submission	21-May-2013
Feedback on Final Submission + Plagiarism Report	28-May-2013
Final Presentation	4-June-2013

Please note that Every Phase has Marks

Present your Work

☺

SURPRISE PRESENTATION

Present your Topic

Steps

- Introduce your self (Your Group)
- Your Topic
- An 2-3min extract of all the papers you have studied
- What are your findings till today?
- Have you finalized your Objectives?

CONTENTS

DESIGN?

- Synonyms: plan, arrangement, lay out, map, scheme
- Antonyms: accident, fluke, chance, guess
- Design is an activity of creating a solution that satisfies a specific goal or need
- Design is the backbone of all products and services
- Considered an artistic and heuristic activity

Software Design - 1

- Software design is an artifact that represents a solution, showing its main features and behavior, and is the basis for implementing a program or collection of programs

- Design is a meaningful representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of pre-defined criteria of "good" design

An Important Point

- Try to associate quality attributes with every aspect of software design

Design and Quality

- Design provides us with representation of software which can be assessed for quality
- Design is the only way that we can accurately translate a customer’s requirements into a finished software product or system

WITHOUT SOFTWARE DESIGN

- We risk building an unstable system
  - one that will fail when small changes are made
  - one that may be difficult to test
  - one whose quality cannot be assessed until late in the software process
  - one that will be of no or very little use for similar projects (not reusable)

Design Process and Model

Software design is both a process and a model

Design Process

- It is a sequence of steps that enables a designer to describe all aspects of the software to be built
- During the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs
- Needs creative skills, past experience, sense of what makes "good" software, and an overall commitment to quality

DESIGN MODEL

- Equivalent to an architect’s plan for a house
- Represents the totality of the thing to be built
- Provides a variety of different views of the computer software

Design Defects

Design Defects - 1

- Defects introduced during preliminary design phase are usually not discovered until integration testing, which is too late in most cases
- Defects introduced during detailed design phase are usually discovered during unit testing

Design Defects - 2

- All four categories of defects are found in design models
  - Errors of commission
  - Errors of omission
  - Errors of clarity and ambiguity
  - Errors of speed and capacity

Design Defects - 3

- Most common defects are errors of omission, followed by errors of commission
- Errors of clarity and ambiguity are also common, and many performance related problems originate in design process also

Design Defects - 4

- Overall design ranks next to requirements as a source of very troublesome and expensive errors
- A combination of defect prevention and defect removal is needed for dealing with design defects

Design Defects - 5

- Formal design inspections are one of the most powerful and successful software quality approaches of all times
- Software professionals should incorporate inspections in their software development process

Defects in Fundamental Design Topics

- Functions performed
- Function invocation, control, and termination
- Data elements
- Data relationships
- Structure of the application
- Sequences or concurrency of execution
- Interfaces

Functions Performed

- Errors in descriptions of functions the application will perform, are often errors of omission
- Often omitted functions are those which, are implied functions, rather than the explicitly demanded functions

Function Invocation, Control, and Termination

- Defects in
  - information on how t start-up a feature
  - control its behavior
  - safely turn off a feature when finished
- are common in commercial and in-house software applications

- Fifty percent of the problems reported to commercial software vendors are of this class

Data Elements

- Errors in describing the data used by the application are a major source of problems downstream during coding and testing
- A minor example of errors due to inadequate design of data elements can be seen in many programs that record addresses and telephone numbers
- Often insufficient space is reserved for names, etc.

Data Relationships

- Errors in describing data relationships are very common and a source of much trouble later

Structure of the Application

- Complex software structures with convoluted control flow tend to have higher error rates
- Poor structural design is fairly common, and is often due to haste or poor training and preparation
- Tools can measure cyclomatic and essential complexity
- Prevention is often better than attempting to simplify an already complex software structure

Sequences or Concurrency of Execution

- Many errors of speed and capacity have their origin in failing to design for optimum performance
- Performance errors are a result of complex control flow, excessive branching, or too many sequential processing (use parallel processing)
- Minimize I/O operations

INTERFACES

- Chronic design problem
- Incompatible data types in message communication

Errors in Eight Secondary Design Topics

- Security
- Reliability
- Maintainability
- Performance
- Human factors
- Hardware dependencies
- Software dependencies
- Packaging

Addressing Design Problems

- Continuously evaluate your design model and design process
- Use design inspections or formal technical reviews, which have proven to be the most valuable mechanism to improve quality of software ever, and especially for design
- Develop software design by following design principles and guidelines

Design Process

Design Process

- The design process produces
  - a data design
  - an architectural design
  - an interface design
  - a component design

Data Design

- The data design transforms the information domain model created during analysis into the data structures that will be required to implement the software

Architectural Design

- It defines the relationship between major structural elements of the software. Architectural design representation is derived from system specification, analysis model, and interaction of subsystems

Interface Design

- Interface design describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information

Component Design

- Component-level design transforms structural elements of software architecture into a procedural description of software components

Think the Right Way

- To achieve a good design, people have to think the right way about how to conduct the design activity
  - Katharine Whitehead

Design Process Principles - 1

- The design process should not suffer from “tunnel vision”
- The design should not reinvent the wheel
- The design should “minimize the intellectual” distance between the software and the problem as it exists in the real world

Design Process Principles - 2

- The design should exhibit uniformity and integration
- The design should be assessed for quality as it is being created, not after the fact
- The design should be reviewed to minimize conceptual (semantic) errors

WHEN APPLYING THEM

- Plan for change, because it is expected
- Plan for failure, because no software system is free of defects

How do we know if the design we have developed is of high quality?

DESIGN PROCESS EVALUATION GUIDE # 1

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer

Design Process Evaluation Guide # 2

- The design must be readable and understandable guide for those who generate code, write test cases, and test the software

DESIGN PROCESS EVALUATION GUIDE # 3

- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective

Design Model/Representation

Design Model Principles - 1

- Separation of concerns
- Modeling real-world objects
- Minimizing the interactions among cohesive design components
- The design should be traceable to the analysis model

Design Model Principles - 2

- The design should be structured to accommodate change
- The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered
- Design is not coding, coding is not design

Guidelines for Good Design Model - 1

- A design should exhibit an architectural structure that
  - Has been created using recognizable design patterns
  - Is composed of components that exhibit good design characteristics
  - Can be implemented in an evolutionary fashion, facilitating implementation and testing

Guidelines for Good Design Model - 2

- A design should be modular; that is software should be logically partitioned into elements that perform specific functions and sub-functions
- The design should contain distinct representations of data, architecture, interfaces, and components (modules)

Guidelines for Good Design Model - 3

- A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns
- A design should lead to components that exhibit independent functional characteristics

Guidelines for Good Design Model - 4

- A design should lead to interfaces that reduce the complexity of connections between modules and with external environment
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis

Questions Answered by Design Concepts

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of software?
- What uniform criteria define the technical quality of a software design?

Abstraction

- Abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low-level details
- Abstraction is one of the fundamental ways that we as humans cope with complexity

Grady Booch

Levels of Abstraction

- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment
- At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution
- At the lowest level of abstraction, the solution is stated in a manner that can be directly implemented

Types of Abstraction

- Procedural abstraction
  - Named sequence of instructions that has a specific and limited function
  - Example: Open door
- Data abstraction
  - Named collection of data that describes a data object
  - Example: any object
- Control abstraction
  - Implies a program control mechanism without specifying internal details
  - Example: synchronization semaphore Variable

Refinement - 1

- A program is developed by successively refining levels of procedural detail
- A hierarchy is developed decomposing a macroscopic statement of function in a step-wise fashion until programming language statements are reached
- Refinement is actually a process of elaboration

Refinement - 2

- There is a tendency to move immediately to full detail, skipping the refinement steps. This leads to errors and omissions and makes the design much mode difficult to review. Perform stepwise refinement
- Abstraction and refinement are complementary concepts

Modularity

- One of the oldest concepts in software design
- Software is divided into separately named and addressable components, often called, modules, that are integrated to satisfy problem requirements
- Modularity is the single attribute of software that allows a program to be intellectually manageable
- Don't over modularize. The simplicity of each module will be overshadowed by the complexity of integration

Quality Design Concepts

INFORMATION HIDING - 1

- Modules should be specified and designed so that information (procedures and data) contained within a module is inaccessible to other modules that have no need for such information
- IH means that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve a software function

Information Hiding - 2

- Abstraction helps to define the procedural (or informational) entities that make up the software
- IH defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module

Cohesion - 1

- Cohesion is the qualitative indication of the degree to which a module focuses on just one thing
- In other words, cohesion is a measure of the relative functional strength of a module
- A cohesive module performs one single task or is focused on one thing
- Highly cohesive modules are better, however, mid-range cohesion is acceptable
- Low-end cohesion is very bad

COUPLING

- Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world
- In other words, coupling is a measure of interconnection among modules in a software structure
- Loose coupling is better. Simple connectivity is easier to understand and less prone to "ripple effect"

Design Methods

- Use a design method, which is most suitable for the problem at hand. Don’t just use the latest or the most popular design method
- There are many structured design and object-oriented design methods to choose from
- Follow the design method’s representation scheme. It helps in understanding design

Programming and Software Quality

PROGRAMMING

- The act of programming, also known as coding, produces the primary products – executables – of a software development effort
- All prior activities culminate in their development
- Programming is done in a programming language

Coding Defects - 1

- All four categories of defects are found in source code
  - Errors of commission
  - Errors of omission
  - Errors of ambiguity and clarity
  - Errors of speed and capacity
- Errors of commission are the most common when the code is underdevelopment

Coding Defects - 2

- The most surprising aspect of coding defects is that more than fifty (50) percent of the serious bugs or errors found in the source code did not truly originate in the source code
- A majority of the so-called programming errors are really due to the programmer not understanding the design or a design not correctly interpreting a requirement

Coding Defects - 3

- Software is one of the most difficult products in human history to visualize prior to having to build it, although complex electronic circuits have the same characteristic
- Built-in syntax checkers and editors with modern programming languages have the capacity to find many “true” programming errors such as missed parentheses or looping problems
- They also have the capacity to measure and correct poor structure and excessive branching

Coding Defects - 4

- The kinds of errors that are not easily found are deeper problems in algorithms or those associated with misinterpretation of design
- At least five hundred (500) programming languages are in use, and the characteristics of the languages themselves interact with factors such as human attention spans and capacities of temporary memory
- This means that each language, or family of languages, tends to have common patterns of defects but the patterns are not the same from language-to-language

Coding Defects - 6

- There is no solid empirical data that strongly-typed languages have lower defect rates than weakly-typed languages, although there is no counter evidence either
- Of course for all programming languages, branching errors are endemic. That is, branching to the wrong location for execution of the next code segment

Defects in High-Level Languages - 1

- Many high-level languages, such as Ada and Modula, were designed to minimize certain common kinds of errors, such as mixing data types or looping incorrect number of times
- Of course, typographical errors and syntactical errors can still occur, but the more troublesome errors have to do with logic problems or incorrect algorithms

DEFECTS IN HIGH-LEVEL LANGUAGES - 2

- A common form of error with both non-procedural and procedural languages has to do with retrieving, storing, and validating data
- It may sometimes happen that the wrong data is requested
- Programming in any language is a complex intellectual challenge with a high probability of making mistakes from time to time
- Analogy with typos in a newspaper

Defects in Low-Level Languages

- Since low-level languages often manipulate registers and require that programmers setup their own loop controls, common errors involve failure to initialize registers or going through loops the wrong number of times, not allocating space for data and subroutines
- For weakly-typed languages, mismatched data types are common errors

Quality Practices for General-Purpose Programming - 1

- Use the highest-level programming language possible
- Use integrated development environments
- Adopt a coding standard that prevents common types of defects

Quality Practices for General-Purpose Programming - 2

- Prototype user interfaces and high-risk components
- Define critical regions

Use the Highest-Level Programming Language - 1

- Code written in higher-level programming languages is easier to read and maintain
- Any fool can write code that a computer can understand. Good programmers write code that humans can understand

Use the Highest-Level Programming Language - 2

- Several practical factors influence the selection of a programming language
  - Technology trends
  - Organizational informational technology strategies
  - Customer restrictions on programming language selection
  - Experience of the development team
  - Features of the programming language (e.g., to interoperate with external systems)

Use the Highest-Level Programming Language - 3

- The complexity of software systems is growing quicker than our ability to develop software solutions
- For example, productivity of computer personnel increased about 6% per year during the 1990s, whereas the growth in NASA mission software is about 25% per year

Use the Highest-Level Programming Language - 4

- Productivity is constant in terms of program statement size. That is writing ten lines of code in assembly language requires as much work as writing ten lines of code in C++, but the functionality developed in ten lines of C++ is much more than the ten lines of assembly language code
- We are shrinking the size of the programs by using higher-level languages

Use the Highest-Level Programming Language - 5

- Fred Brooks has said that the advent of high-level programming languages had the greatest impact on software productivity because there was at least a factor of five improvement in productivity
- The use of high-level programming languages results in more reliable software

Use Integrated Development Environments

- Also known as IDEs, these suites include an editor, a compiler, a make utility, a profiler, and a debugger. Other tools may also be included
- Recent IDEs include tools to model software designs and implement graphical user interfaces
- These tools, if used properly, can improve the productivity 100%
- They also help identify many coding defects, as they are being introduced in the software

ADOPT A CODING STANDARD TO PREVENT COMMON TYPES OF DEFECTS - 1

- Coding standards are controversial because the choice among many candidate standards is subjective and somewhat arbitrary
- Standards are most useful when they support fundamental programming principles

Adopt a Coding Standard to Prevent Common Types of Defects - 2

- So, it is easier to adopt a standard for handling exceptions, than for identifying the amount of white-space to use for indentation
- An organization should always ask itself whether a coding standard improves program comprehension characteristics

Quality Practices Related to Programming

Practices for Internal Documentation - 1

- Specify the amount of white-space that should be used and where it should appear
  - Before and after loop statements and function definitions
  - At each indentation level (two or four spaces have been reported as improving comprehensibility of programs)
- Physically offset code comments from code when contained on the same line

Practices for Internal Documentation - 2

- Use comments to explain each class, function, and variable contained in source code. (Comments can be from 10% and up)
  - Key interactions that a function has with other functions and global variables
  - Complex algorithms used by every function
  - Exception handling
  - Behavior and effect of iterative control flow statements and interior block statements

Practices for Internal Documentation - 3

- Provide working examples in the user documentation or tutorial materials

Practices for Variable Definition - 1

- Declare variables as specifically as possible and initialize them, preferably one declaration per line
- Do not use similarly named variables within the same lexical scope
- Consistently, use clear and easily remembered names for variables, classes, and functions

Practices for Variable Definition - 2

- Follow a uniform scheme when abbreviating name
- Do not use local declarations to hide declarations at greater scope
- Never use a variable for more than one purpose

Practices for Control Flow

- Do not assume a default behavior for multi-way branches
- Do not alter the value of an iteration variable within a loop
- Use recursion, when applicable

Practices for Functions

- Explicitly define input and output formal parameters
- Use assertions (e.g., pre- and post-conditions) to verify the accuracy and correctness of input and output formal parameters. The use of pre- and post-conditions helps programmers detect defects closer to their origin

Practices for Operations

- Make all conversion of data explicit, especially numeric data
- Do not use exact floating-point comparison operations
- Avoid using operators in potentially ambiguous situations

Practices for Exception Handling

- Process all exceptions so that personnel can more easily detect their cause
- Log important system events, including exceptions

Practices for Maintenance

- Isolate the use of nonstandard language functions
- Isolate complex operations to individual functions

Practices for Operational

- Do not permit any compilation to produce warnings
- Optimize software only after it works is complete, and only if required to achieve performance goals

Prototype User Interfaces and High-Risk Components

- User interface prototyping helps identify necessary features that software engineers might otherwise overlook
- Prototyping can reduce the development effort significantly
- Prototyping reduces development risk because is allows programmers to explore methods for achieving performance and other high-risk requirements

Define Critical Regions

- A task that interrupts an interdependent operational sequence before it is completed can leave a program in a vulnerable state, resulting in inconsistent and inaccurate results. We need a critical regions to run such transactions
- Critical regions help prevent deadlocks

References

- Software Engineering Quality Practices by Ronald K. Kandt (Ch. 8)
- Software Quality: Analysis and Guidelines for Success by Capers Jones
- Software Quality: Analysis and Guidelines for Success by Capers Jones
- Software Engineering: A Practitioner's Approach by Roger Pressman (Chapter 13)
- Software Engineering Quality Practices by Ronald K. Kandt

Thanks!
mc