Software Engineering
CS5704: Class 11 - 4/6/01

Instructor: Shawn A. Bohner
Voice: (703) 538-8374
Email: bohner@nvc.cs.vt.edu
Teaching Assistant: Sapna Georges
Voice: (703) 538-8381
Email: sgeorge@vt.edu

Agenda

▲ Turn in Project Team Results
▲ Class 10 Review
  ● Chapter 22 – Object-Oriented Design
  ● Chapter 14 – Architectural Design
  ● Chapter 15 – User Interface Design
  ● Chapter 16 – Component Level Design

▲ Chapter 17 – Testing Techniques
Break
▲ Chapter 18 – Testing Strategies
▲ Chapter 23 – Object-Oriented Testing
▲ Homework Assignments
Spring Semester Timeline

Class Begins Product & Process
PM Metrics & Estimation
Midterm Exam
Analysis, Design, & Architecture
SW Metrics & Testing Strategies
Maintenance & Evolution
Final Exam

Jan — Feb — Mar — Apr — May
Project Management
Cross-Life-Cycle Process
Testing Techniques
Object-Oriented Development
Advanced SWE Topics

4 weeks, 4 sessions to go... we’re in the home stretch... Still, so much to do and so little time...

OOA and OOD

Attributes, operations, collaborators
CRC Index Cards
Object-relationship model
Use cases
ObjectBehavior Model
responsibilities design
message design
Class and object design
subsystem design

How is this different from Structured Analysis Structured Design?
More OOA and OOD

Analysis Model
- classes
- attributes
- methods
- relationships
- behavior

Design Model
- ???
- ???
- ???
- ???
- ???

Process Flow for OOD

- System Design
- Object-Oriented Analysis
- Task Management Design
- Data Management Design
- Human Interface Design
- Object Design
Why Architecture?

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

(1) analyze the ??? of the design in meeting its stated requirements,

(2) consider ??? alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the ??? associated with the construction of the software.

Architectural Styles

Each style describes a system category that encompasses:

1) a set of components
2) a set of connectors
3) constraints, and
4) semantic models.

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures
Structured Design

- **Objective:** to derive a partitioned program architecture.
- **Approach:**
  - the DFD is mapped into a program architecture
  - the PSPEC and STD are used to indicate the content of each module
- **Notation:** Structure

Second Level Mapping

Which one is the structure chart?
Why is it important to map to the program design?
## Partitioning the Architecture

- What does Program Architecture Achieve?
  - Software that is easier to ...

- What is vertical partitioning?

- What is horizontal partitioning?

![Diagram of partitioned architecture]

## User Interface Design

<table>
<thead>
<tr>
<th>Easy to learn?</th>
<th>Easy to use?</th>
<th>Easy to understand?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Golden Rules

- Place the user in control
- Reduce user’s memory load
- Make the interface consistent

### Typical Design Errors

- Lack of ???
- Too much ???
- No ???
- No ??? sensitivity
The Component-Level Design Model

▲ Represents the ??? at a level of detail that can be reviewed for quality
▲ Options:
  • Graphical (e.g. ???)
  • Pseudocode (e.g., ???)
  • Programming language (e.g., ???)
  • Decision table (e.g., ???)

Chapter 17: SW Testing Techniques

▲ The purpose of this chapter is to provide an overview of the software testing techniques.
  • Approaches for developing software test cases
  • Black-box and white-box testing techniques
▲ Objectives
  • Present an introduction to software testing.
  • Outline key software testing techniques
  • Examine key test technique examples.
What is Software Testing?

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

In other words, testing is about “increasing the confidence” that the delivered software does not contain defects that will not cause failures during operation.

Testing Across the Software Process
Testing Reality Check...

▲ Testing is about confidence...
  • Testing conducted to “assess reliability” or “detect defects”
  • Testing can only show the existence of defects, not their absence

▲ We often chase the wrong bugs...
  • 1/3 of all defects fail less than once every 5 execution years

▲ Costs to develop software are proportional to the costs of testing
  • Limited by time and cost constraints instead of by quality thresholds

Testing Practice Today

▲ Most common testing techniques in use today were developed in the late 1960’s
  • What if your doctor used only surgery techniques s/he learned 40 years ago?

▲ More advanced techniques (e.g., instrumentation, design for test, mutant testing, etc.) are:
  • Used by a testing specialist groups
  • Growing among firms with large IT portfolios where economy of scale applies
McCall’s Triangle of Quality

Maintainability
Flexibility
Testability

Portability
Reusability
Interoperability

Correctness
Usability
Efficiency

Reliability
Integrity

Testability Factors

▲ Operability—it operates cleanly
▲ Observability—the results of each test case are readily observed
▲ Controlability—the degree to which testing can be automated and optimized
▲ Decomposability—testing can be targeted
▲ Simplicity—reduce complex architecture and logic to simplify tests
▲ Stability—few changes are requested during testing
▲ Understandability—of the design

Do these sound like requirements factors?
Who Tests the Software?

**Developer**
- Understands the system
- Will test "gently"
- Driven by "delivery"

**Independent Tester**
- Must learn about the system,
- Will attempt to break it
- Driven by quality

Is Exhaustive Testing Cost Effective?

There are $10^{14}$ possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!
Is Selective Testing Sufficient?

Software Testing is a confidence exercise…
When is enough, enough?

Test Case Design

"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer

OBJECTIVE       To uncover errors / defects
CRITERIA        In as complete manner as possible
CONSTRAINT      With a minimum of effort and time
White-Box Testing

Why Coverage?
- Logic errors and incorrect assumptions are inversely proportional to a path’s execution probability
- We often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
- Typographical errors are random; it’s likely that untested paths will contain some

*Good for critical modules*

Basis Path Testing

- First compute cyclomatic complexity -- V(G)
  - # of simple decisions + 1
  - # of enclosed areas + 1
- Derive the independent paths
  - V(g) = 4, thus 4 paths possible
  - Path 1: 1,2,3,6,7,8
  - Path 2: 1,2,3,5,7,8
  - Path 3: 1,2,4,7,8
  - Path 4: 1,2,4,7,2,4,...,7,8
- Derive test cases to exercise these paths
Loop Testing: Simple & Concatenated Loops

**Minimum conditions**

1. Skip the loop entirely
2. Only one pass through the loop
3. Two passes through the loop
4. \(m\) passes through the loop \(m < n\)
5. \((n-1), n,\) and \((n+1)\) passes through the loop

where \(n\) is the maximum # of allowable passes

- If concatenated loops are independent then treat each as a simple loop
- Otherwise treat them as nested loops

Loop Testing: Nested & Unstructured Loops

⚠️ **Start at the innermost loop**
- Set all outer loops to their minimum iteration parameter values
- Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values

⚠️ **Move out one loop and set it up as in step 2, holding all other loops at typical values**
- Continue this step until the outermost loop has been tested

⚠️ **If unstructured, simplify**
Black-Box Testing

Requirements → Input → Events → Output

Equivalence Partitioning

**Valid Data**
- User Supplied Commands
- Responses to System Prompts
- File Names
- Computational Data
  - Physical Parameters
  - Bounding Values
  - Initiation Values
- Output Data Formatting
- Responses to Error Messages
- Graphical Data (E.g., Mouse Picks)

**Invalid Data**
- Data Outside Bounds of the Program
- Physically Impossible Data
- Proper Value Supplied in Wrong Place
Chapter 18: Software Testing Strategies

- The purpose of this chapter is to introduce a strategic software testing approach applicable to most software development projects
  - Testing must be planned and assessed for quality like any other SW engineering process
  - Testing process includes unit testing, integration testing, system testing, and acceptance testing
  - Difference between testing and debugging

- Objectives
  - Introduce to software testing process.
  - Outline key software testing strategies and principles
Software Testing Strategy

- **Acceptance Testing**
  - Plan
  - Develop
  - Execute

- **System Testing**
  - Plan
  - Develop
  - Execute

- **Integration Testing**
  - Plan
  - Develop
  - Execute

- **Unit Testing**
  - Plan
  - Develop
  - Execute

**Unit Testing**

- Software Engineer
  - Module to be Tested
  - Results
  - Interface
  - Local Data Structures
  - Boundary Conditions
  - Independent Paths
  - Error Handling Paths
  - Test Cases
Unit Test Environment

- Driver
- Results
- Module
- Stub
- Stub
- Interface
- Local Data Structures
- Boundary Conditions
- Independent Paths
- Error Handling Paths
- Test Cases

Integration Testing Strategies

Options:
- The “big bang” approach
- An incremental construction strategy
Top Down Integration

Top Module is Tested With Stubs

Stubs Are Replaced One at a Time, "Depth First"

As New Modules are Integrated, Some Subset of Tests is Re-run

Bottom-Up Integration

Drivers Are Replaced One at a Time, "Depth First"

Worker Modules Are Grouped Into Builds and Integrated
Sandwich Testing

Top modules are tested with stubs

Worker modules are grouped into builds and integrated

Cluster

System Testing

- Process of testing an integrated system to verify that it meets its specified requirements from a user's point of view
  - Objective is to develop confidence in the readiness for acceptance testing
  - Assures the "system" does what it was designed to do under use conditions
  - Includes load, stress, & performance testing
- Ideally an independent tester conducts system testing since they represent the user
  - They should perform the concurrent tasks with the software staff; system testers should develop an environment to test the system
  - This is of particular value in the regression testing suite for future testing
Acceptance Testing

- Acceptance testing entails conducting formal tests to determine whether or not a system satisfies the acceptance criteria negotiated with the customer
  - This enables the customer to determine whether or not to accept the system
  - The customer or an authorized customer representative conducts acceptance testing

- Planning for acceptance testing begins with requirements specification
- Execution of acceptance testing starts when system testing is complete

Regression Testing

- Regression Testing is the re-execution of tests that have already been used to ensure changed have not propagated unwanted side effects
  - Employs a suite of test cases to ensure outputs from the original code and changed code are the same where no change is needed

- Tools that support regression testing primarily manage test cases and suites
  - Enabling staff members to determine which tests have been executed and compare the results
Application Testing In-the-Large

- Supports the systems of systems view
- While development testing is frequently compressed due to schedule pressure, production and operational testing is typically ad hoc for most IT shops
- Software and computer producers have better practices for testing in-the-large

Advice: Never test an error you don’t know how to handle!

Packaged Applications Testing

- Testing Packaged Applications takes place largely at the system level (black-box)
- System-level testing seldom indicates how the system behaves when a specific component fails
  - Very few packages support instrumentation to support testing
  - Tracing back to the source is nearly impossible
- A more rigorous approach may be needed
Software Fault Injection

- Interface Propagation
  - What-if analysis
  - not statistical testing
  - not correctness proof

- More what-if gaming, more confidence

- Augment with a vendor strategy that gets developer to warranty against failure classes

Wrapping Before You Test

- Prevent package components from certain functions

- Two Goals
  - Removing functionality
  - Recovery

- Tends to be complex and error prone... but a plausible alternative

Don’t forget to test the wrapper!
Testing a Package Only Solution

- If B is correct, but the interface between B and C is faulty, will that affect the system?
- If B is correct, but the interface between B and A is faulty, will that affect the system?
- If B is faulty, will its errors alter behavior of the system and if so, how?
- Can B exacerbate problems in A or C that have so far not affected the system?

Debugging: A Diagnostic Process

- Test Inputs
- Outputs???

- Test Cases
- Regression Tests
- Suspected Causes
- Identified Causes
- New Test Cases
- Results

- Debugging: Brute Force / Testing
- Backtracking
- Induction
- Deduction
Symptoms & Causes

- Symptom and Cause may be Geographically Separated
- Symptom may Disappear when Another Problem is Fixed
- Cause may be Due to a Combination of Non-errors
- Cause may be Due to a System or Compiler Error
- Cause may be Due to Assumptions that Everyone Believes
- Symptom may be Intermittent

Consequences of Bugs

Entomological Categories: function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.
Chapter 23: Object-Oriented Testing

The purpose of this chapter is to examine object-oriented testing (OOT) in light of conventional software testing
- OOT focuses on the OO class, rather than individual operations within a class
- Lowest level of OOT resembles first level of integration testing in conventional testing
- OOT begins during review of OOA/D models

Objectives
- Introduce key OOT principles through comparison with conventional software testing
- Illustrate benefits of OOT through examples

Object-Oriented Testing

Begin by evaluating the correctness and consistency of the OOA and OOD models

Testing Strategy Changes
- Concept of “Unit” broadens due to encapsulation
- Integration focuses on classes and their execution across a ‘thread’ or in the context of a usage scenario
- System/Validation uses conventional black box methods

Test case design draws on conventional methods, but also encompasses some special features
Broadening the View of “Testing”

The review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level.

Problems in the definition of class attributes uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

Testing the CRC Model

1. Revisit the CRC model and the object-relationship model.

2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator’s definition.

3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.

4. Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.

5. Determine whether widely requested responsibilities might be combined into a single responsibility.

6. Steps 1 to 5 are applied iteratively to each class and through each evolution of the OOA model.
OOT Strategy

▲ Class testing is like unit testing
- Operations within the class are tested
- State behavior of the class is examined

▲ 3 Integration testing strategies
- Thread testing—integrates the set of classes required to respond to one input or event
- Use-based testing—integrates the set of classes required to respond to one use case
- Cluster testing—integrates the set of classes required to demonstrate one collaboration

OOT—Test Case Design

Ed Berard proposes the following approach:

1. Each test case should be uniquely identified and should be explicitly associated with the class to be tested
2. The purpose of the test should be stated
3. A list of testing steps should be developed for each test and should contain [BER94]:
   a. a list of specified states for the object that is to be tested
   b. a list of messages and operations that will be exercised as a consequence of the test
   c. a list of exceptions that may occur as the object is tested
   d. a list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)
   e. supplementary information that will aid in understanding or implementing the test.
OOT Methods: Random Testing

▲ Identify operations applicable to a class
▲ Define constraints on their use
▲ Identify a minimum test sequence
  • An operation sequence that defines the minimum life history of the class (object)
▲ Generate a variety of random (but valid) test sequences
  • Exercise other (more complex) class instance life histories

OOT Methods: Partition Testing

▲ Like equivalence partitioning, it reduces the number of test cases required to test a class
▲ State-Based Partitioning
  • Categorize / test operations based on their ability to change the state of a class
▲ Attribute-Based Partitioning
  • Categorize / test operations based on the attributes that they use
▲ Category-Based Partitioning
  • Categorize and test operations based on the generic function each performs
OOT Methods: Inter-Class Testing

▲ For each client class, use the list of class operators to generate a series of random test sequences.
  • The operators will send messages to other server classes.
▲ For each message that is generated, determine the collaborator class and the corresponding operator in the server object.
▲ For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
▲ For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence

Homework Assignment for 4/13/01

▲ Read Pressman Chapters
  • Chapter 19 – Technical Metrics for Software
  • Chapter 24 – Technical Metrics for Object-Oriented Systems
▲ “Dog E-Dating System” Project
  • Continue to refine detailed Object-Oriented Design
  • Start Overall Test Plan with Test Cases for System and Integration Testing
  • Final Project Package Due April 20th, 2001
▲ Have a great week!