Chapter 14: Testing (2/2)

Object-Oriented Software Construction

Armin B. Cremers, Sascha Alda & Tobias Rho
(based on Bruegge & Dutoit)
Types of Testing

♦ Unit Testing (last lecture):
   ♦ Individual subsystem
   ♦ Carried out by developers (of components)
   ♦ Goal: Confirm that subsystems is correctly coded and carries out the intended functionality

♦ Integration Testing (mainly this lecture):
   ♦ Groups of subsystems (collection of classes) and eventually the entire system
   ♦ Carried out by developers
   ♦ Goal: Test the interface and the interplay among the subsystem
Types of Testing

♦ System Testing:
  ♦ The entire system
  ♦ Carried out by developers (testers!)
  ♦ Goal: Determine if the system meets the requirements (functional and global)
  ♦ Functional Testing: Test of functional requirements
  ♦ Performance Testing: Test of non-functional requirements

♦ Acceptance and Installation Testing:
  ♦ Evaluates the system delivered by developers
  ♦ Carried out by the client.
  ♦ Goal: Demonstrate that the system meets customer requirements and is ready to use
Integration Testing Strategy

♦ The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.

♦ Assumption: System Decomposition is hierarchical

♦ The order in which the subsystems are selected for testing and integration determines the testing strategy
  ♦ Big bang integration (Nonincremental)
  ♦ Bottom up integration
  ♦ Top down integration
  ♦ Sandwich testing
  ♦ Variations of the above

♦ For the selection use the system decomposition from the System Design
Example: Three Layer Call Hierarchy
Integration Testing: Big-Bang Approach

- All components (units) are first tested individually and then together as a single and entire system:

- Pros:
  - No test stubs (mocks) and drivers are needed

- Cons:
  - Difficult to pinpoint the specific component responsible for the failure

⇒ Results in Strategies that integrate only a few components at the time

Don’t try this!
Bottom-up Testing Strategy

- The subsystem in the lowest layer of the call hierarchy are tested individually.
- Then the next subsystems are integrated and tested from the next layer up that call the previously tested subsystems.
- This is done repeatedly until all subsystems are included in the testing.
- Only Test Drivers are used to simulate the components of higher layers.
- No Test Stubs!
Bottom-up Integration

Test E
Test F
Test C
Test G
Test B, E, F
Test D, G
Test A, B, C, D, E, F, G
Pros and Cons of bottom up integration testing

♦ Pros:

♦ Interface faults can be more easily found (the usage of test drivers accomplishes a clear intention of the underlying interfaces of the lower layer)
♦ No Stubs are necessary

♦ Cons:

♦ Components of the User Interface are tested last
♦ Faults found in the top layer may lead to changes in the subsystems of lower layers, invalidating previous tests.
Top-down Testing Strategy

♦ Test the top layer or the controlling subsystem first
♦ Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
♦ Do this until all subsystems are incorporated into the test
♦ Test Stubs are used to simulate the components of lower layers that have not yet been integrated.
♦ No drivers are needed
Top-down Integration Testing

Layer III

Test A → Test A, B, C, D → Test A, B, C, D, E, F, G

Layer III + II

All Layers
Pros and Cons of top-down integration testing

♦ Pros:
  ♦ Test cases can be defined in terms of the functionality of the system (functional requirements)
  ♦ More effective for finding faults that are visible to the user

♦ Cons:
  ♦ Writing stubs can be difficult: Stubs must allow all possible conditions to be tested.
  ♦ Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.
Sandwich Testing Strategy

♦ Combines top-down strategy with bottom-up strategy (parallel testing is possible)

♦ The system is view as having three layers
  ♦ A target layer in the middle
  ♦ A layer above the target (top layer)
  ♦ A layer below the target (bottom layer)
  ♦ Testing converges towards the target layer

♦ No Test Stubs and Drivers are necessary for bottom and top layer
Sandwich Testing Strategy

Bottom Layer Tests

- Test E
- Test F
- Test G

Top Layer Tests

- Test A
- Test B, E, F
- Test D, G
- Test A, B, C, D

Test A, B, C, D, E, F, G

Layer III

Layer I

Layer I
Pros and Cons of Sandwich Testing

♦ Pros:
  ✷ Top and Bottom Layer Tests can be done in parallel
  ✷ No Stubs and Drivers (saves development time)

♦ Cons:
  ✷ Does not test the individual subsystems on the target layer thoroughly before integration (C in the example)

♦ Solution: Modified sandwich testing strategy
Modified Sandwich Testing Strategy

♦ Tests the three layers individually before combining them in incremental tests with one another

♦ The individual layer tests consists of three tests:
  ◆ Target layer test with drivers and stubs
  ◆ Top layer test with stubs
  ◆ Bottom layer test with drivers

♦ The combined Layer Tests consist of two tests:
  ◆ Top layer accessing target layer (top layer replaces drivers)
  ◆ Bottom accessed by target layer (bottom layer replaces stubs)
Modified Sandwich Testing Strategy

Layer III

Layer II

Layer I

Test A

Test B

Test C

Test D

Test E

Test F

Test G

Test A, B, C, D

Test A, B, C, D

Test B, E, F

Test D, G

Test A, B, C, D, E, F, G
Using the Bridge Pattern to enable early Integration Testing

- Use the bridge pattern to provide multiple implementations under the same interface.
- Interface to a component that is incomplete, not yet known or unavailable during testing
Which Integration Strategy should you use?

♦ Factors to consider
  ♦ Location of critical parts in the system
  ♦ Availability of hardware
  ♦ Availability of components
  ♦ Scheduling concerns

♦ Bottom up approach
  ♦ good for object oriented design methodologies
  ♦ Test driver interfaces must match component interfaces
  ♦ ...

♦ Top down approach
  ♦ Test cases can be defined in terms of functions examined
  ♦ Need to maintain correctness of test stubs
  ♦ Writing stubs can be difficult

...Top-level components are usually important and cannot be neglected up to the end of testing

♦ Detection of design errors postponed until end of testing

Top-level components are usually important and cannot be neglected up to the end of testing

Detection of design errors postponed until end of testing
System Testing

♦ Functional Testing
♦ Performance Testing
♦ Acceptance Testing
♦ Installation Testing

Impact of requirements on system testing:

♦ The more explicit the requirements, the easier they are to test.
♦ Quality of use cases determines the ease of functional testing
♦ Quality of nonfunctional requirements and constraints determines the ease of performance tests:
Functional Testing

♦ Functional testing finds differences between functional requirements and the implemented system
♦ Essentially the same as black box testing
♦ Goal: Test functionality of system
♦ Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
♦ Select tests that are relevant to the user and have a high probability of uncovering a failure
  ♦ Use techniques like equivalence tests
Performance Testing

♦ Stress Testing
  ♦ Checks if the system can respond to many simultaneous requests (maximum # of users, peak demands)

♦ Volume testing
  ♦ Test what happens if large amounts of data are handled

♦ Configuration testing
  ♦ Test the various software and hardware configurations

♦ Compatibility test
  ♦ Test backward compatibility with existing systems

♦ Security testing
  ♦ Try to violate security requirements

♦ Timing testing
  ♦ Evaluate response times and time to perform a function

♦ Environmental test
  ♦ Test tolerances for heat, humidity, motion, portability

♦ Quality testing
  ♦ Test reliability, maintainability & availability of the system

♦ Recovery testing
  ♦ Tests system’s response to presence of errors or loss of data.

♦ Human factors testing
  ♦ Tests user interface with user
Test Cases for Performance Testing

♦ Push the (integrated) system to its limits.
♦ Goal: Try to break the subsystem
♦ Test how the system behaves when overloaded.
  - Can bottlenecks be identified? (First candidates for redesign in the next iteration)
♦ Try unusual orders of execution
  - Call a receive() before send()
♦ Check the system’s response to large volumes of data
  - If the system is supposed to handle 1000 items, try it with 1001 items.
♦ What is the amount of time spent in different use cases?
  - Are typical cases executed in a timely fashion?
Steps in Testing (only developer’s view)

1. Based on the integration strategy, select a component to be tested. Unit test all the classes in the component.

2. Put selected component together; do any preliminary fix-up necessary to make the integration test operational (drivers, stubs)

3. Do functional testing: Define test cases that exercise all uses cases with the selected component

4. Execute performance tests

5. Keep records of the test cases and testing activities.

6. Repeat steps 1 to 5 until the full system is tested.

The primary goal of integration testing is to identify errors in the (current) component configuration.
Acceptance Testing

♦ Goal: Demonstrate system is ready for operational use
  ♦ Choice of tests is made by client
  ♦ Many tests can be taken from integration testing
  ♦ Acceptance test is performed by the client, not by the developer.
♦ Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:

♦ Alpha test:
  ♦ Sponsor uses the software at the developer’s site.
  ♦ Software used in a controlled setting, with the developer always ready to fix bugs.

♦ Beta test:
  ♦ Conducted at sponsor’s site (developer is not present)
  ♦ Software gets a realistic workout in target environment
  ♦ Potential customer might get discouraged
Summary

♦ Testing is still a black art, but many rules and heuristics are available

♦ Testing consists of component-testing (unit testing, integration testing) and system testing

♦ Design Patterns can be used for integration testing

♦ Testing has its own lifecycle