Unit 9. System & Integration Testing

1. Introduction
2. Integration Testing
3. System Testing Overview
4. System Testing – Testing Functionality
5. System Testing – Global Properties

Reading: TB-Chapter 7 (7.1-7.4), 8 (8.1)

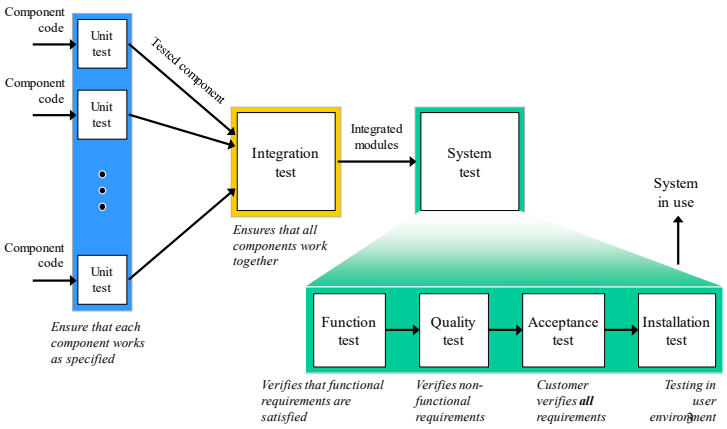
1. Introduction

-A ***system*** is composed of ***components***. System of software components can be defined at any physical scope.

Component (Focus of Integration)	System (Scope of Integration)	Typical inter-component interfaces (locus of integration faults)
Method	Class	Instance variables Intraclass messages
Class	Cluster	Interclass messages
Cluster	Subsystem	Interclass messages Inter-package messages
Subsystem	System	Inter-process communication Remote procedure call ORB services, OS services

Logical Organization of Testing

(Not necessarily how it's actually done!)



Integration Testing Scope

-***Integration testing*** is a search for component faults that cause inter-component failures.

-It is the phase in software testing in which individual software modules are combined and tested as a group.

- Occurs after **unit testing** and before **system testing**.

System Scope Testing

-***System scope testing*** is a search for faults that lead to a failure to meet a system scope responsibility.

•System scope testing cannot be done unless components interoperate sufficiently well to exercise system scope responsibilities.

-Effective testing at system scope requires a concrete and testable system-level specification.

2. Integration Testing

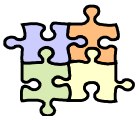
-***Unit testing*** focuses on individual components. Once faults in each component have been removed and the test cases do not reveal any new fault, components are ready to be integrated into larger subsystems.

-***Integration testing*** detects faults that have not been detected during unit testing, by focusing on small groups of components.

•Two or more components are integrated and tested, and once tests do not reveal any new faults, additional components are added to the group.

•The ***order in which components are tested*** can influence significantly the total effort required by the integration test.

•A careful ordering of components can reduce the overall resources needed for the overall integration test.



Integration Faults

- Inconsistent interpretation of parameters or values
  - Example: Mixed units (meters/yards) in Martian Lander
- Violations of value domains, capacity, or size limits
  - Example: Buffer overflow
- Side effects on parameters or resources
  - Example: Conflict on (unspecified) temporary file
- Omitted or misunderstood functionality
  - Example: Inconsistent interpretation of web hits
- Nonfunctional properties
  - Example: Unanticipated performance issues
- Dynamic mismatches
  - Example: Incompatible polymorphic method calls

Integration Testing Strategies

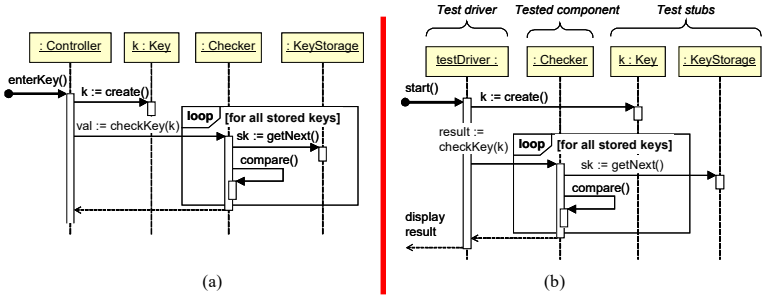
-Several approaches have been devised to implement an integration testing strategy:

- Big bang testing (*when target small or only a few new components are added*)
- Bottom-up testing (*most widely used*)
- Top-down testing (*instances such as façade pattern implementations*)
- Sandwich testing

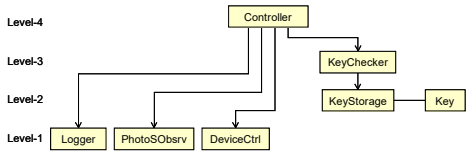
-All these strategies assume originally a hierarchical decomposition of the system, although they can easily be adapted to non-hierarchical system decompositions.

Integration Testing Strategies (ctd.)

Example: Testing a Key Checker



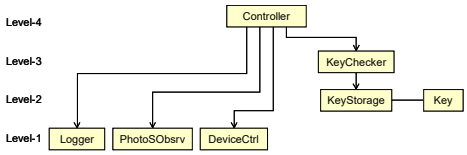
System hierarchy:



Integration Testing Strategies (ctd.)

Big bang testing strategy

System hierarchy:



- Assumes that all components are ***first tested individually*** and ***then tested together*** as a single system.
  - **Advantage:** ***no additional test stubs*** or drivers are needed.
  - **Disadvantage:**
    -Although it sounds simple, it can be ***very expensive***, because if a test uncovers a failure, it is difficult to pinpoint the specific component (or combination of components) responsible for the failure.
    -In addition, it is ***impossible to distinguish failures in the interface from failures within a component.***

Top-down Testing Strategy

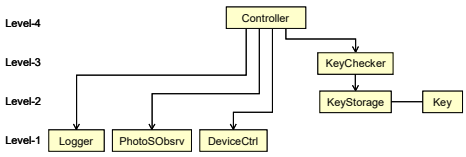
-***Unit test the components of the top layer first*** and then ***integrate the components of the next layer down***. This is ***repeated*** until all layers are combined and involved in the test.

-**Test stubs are used** to simulate the components of lower layers that have not yet been integrated. (Test drivers are not needed in top-down testing).

-**Advantage**: it ***starts with important components such as UI.***

-**Disadvantage**: a ***large number of stubs is usually required***, and the development of ***stubs is time consuming and prone to error.***

System hierarchy:



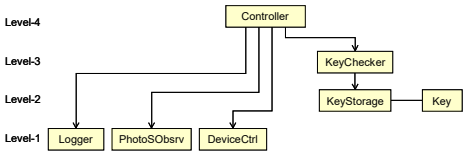
Top-down integration testing:

Bottom-up Testing Strategy

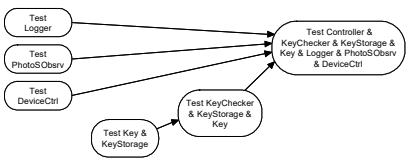
-First ***individually test each component*** of the bottom layer and then ***integrate them with components of the next layer up***. This is ***repeated*** until all components from all layers are integrated.

-**Test drivers are used** to simulate the components of higher layers that have not yet been integrated. (Test stubs are not needed in bottom-up testing).

System hierarchy:



Bottom-up integration testing:



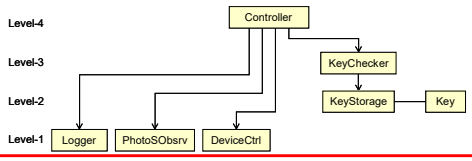
Sandwich Testing Strategy

-***Combine top-down and bottom-up*** strategies, attempting to make use of the best of both strategies.

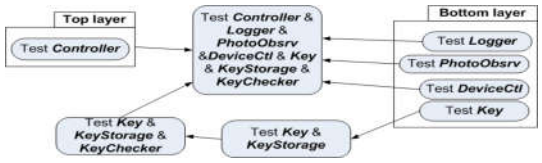
-Reformulate or map the subsystem decomposition into three layers: a target layer ("the meat"), a layer above the target layer ("the bread above"), and a layer below the target layer ("the bread below").

-Using the target layer as the focus of attention, top-down testing and bottom-up testing are conducted in parallel.

System hierarchy:



Sandwich integration strategy:



3. System Testing Overview

-Unit and integration tests focus on finding faults in individual components and the interfaces between them.

-Once components have been integrated, system testing ensures that the complete system complies with the functional and nonfunctional requirements of the system.

-There are various categories of system testing, including:

- Functional testing
- Performance testing
- Security testing
- Recovery testing
- Acceptance testing
- Pilot testing
- Installation testing

System Testing Overview (cont.)

• Key characteristics:

– Comprehensive (the whole system, the whole spec)
– Based on
  • specification of observable behavior
  • verification against a requirements specification, not validation, and not opinions

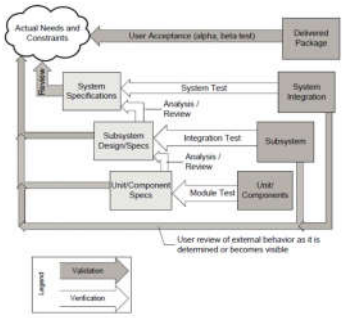
– Independent of design and implementation
  • *Independence*: Avoid repeating software design errors in system test design

System Testing Overview (cont.)

• Develop system test cases early
– As part of requirements specification, before major design decisions have been made

• Agile "test first" and conventional "V model" are both examples of designing system test cases before designing the implementation

• An opportunity for "design for test": structure system for critical system testing early in project



System Testing Overview (cont.)

• Incremental System Testing

- System tests are often used to measure progress
  • System test suite covers all features and scenarios of use
  • As project progresses, the system passes more and more system tests

- Assumes a "threaded" incremental build plan: Features exposed at top level as they are developed

4. System Testing – Testing Functionality

-Functional testing ***tracks differences between the functional requirements and the system***.

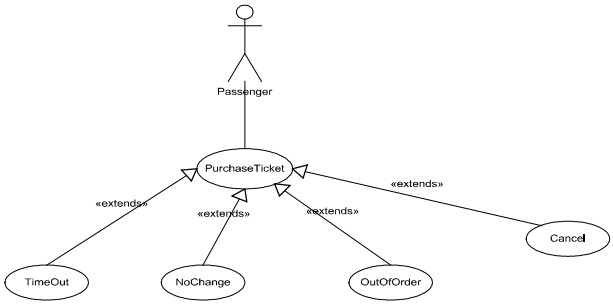
•System testing is a black box testing technique: test cases are typically derived from the **use case model**.

•In systems with complex functional requirements, it is usually not possible to test all use cases for all valid and invalid inputs. The goal of the tester is to select those tests that are relevant to the user and have high probability of uncovering a failure.

•Functional testing is different from usability testing, which also focuses on the use case model.
-Functional testing finds differences between the use case model and observed system behavior;
-Usability testing finds differences between the use case model and the user's expectation of the system.

-To identify functional tests, we inspect the use case model and identify use case instances that are likely to cause failures.
•This is done using black box techniques similar to equivalence and boundary testing.
•Test cases should exercise both common and exceptional scenarios.

Example: Use Case Model for a subway ticket distributor



Flow of Events for PurchaseTicket Use Case

Precondition: The Passenger is standing in front of ticket Distributor
The Passenger has sufficient money to purchase ticket.

Main Flow of Events:

1. The Passenger selects the number of zones to be traveled. If the Passenger presses multiple zone buttons, only the last button pressed is considered by the Distributor.
2. The Distributor displays the amount due.
3. The Passenger inserts money.
4. If the Passenger selects a new zone before inserting sufficient money, the Distributor returns all the coins and bills inserted by the Passenger.
5. If the Passenger inserted more money than the amount due, the Distributor returns excess change.
6. The Distributor issues the ticket.
7. The Passenger picks up the change and the ticket.

Postcondition: The Passenger has the selected ticket.

Example: Test Ideas for PurchaseTicket use case

-We notice that three features of the Distributor are likely to fail and should be tested:

1. The Passenger may press multiple zone buttons before inserting money, in which case the Distributor should display the amount of the last zone.
2. The Passenger may select another zone button after beginning to insert money, in which case the Distributor, should return all money inserted by the Passenger.
3. The Passenger may insert more money than needed, in which case the Distributor should return the correct change.

-Based on the above consideration, we can devise appropriate test cases by describing corresponding flow of events, as well the inputs to the system and the desired or expected outputs.

Example Test Case derived from PurchaseTicket use case

Test case name	**PurchaseTicket_CommonCase**
Entry condition	The Passenger standing in front of ticket Distributor The Passenger has one $5 bill, one $1 bill and three quarters
Flow of events	1. The Passenger presses in succession the zone buttons 2, 4, 1, and 2. 2. The Distributor should display in succession $1.25, $2.25, $0.75, and $1.25 3. The Passenger inserts a $5 bill. 4. The Distributor returns three $1 bills and three quarters and issues a 2-zone ticket 5. The Passenger repeats steps 1-3 using five quarters. The Distributor issues a 2-zone ticket. 6. The Passenger selects zone 1 and inserts a dollar bill. The Distributor issues a 1-zone ticket and returns a quarter. 7. The Passenger selects zone 4 and inserts two $1 bill and a quarter. The Distributor issues a 4-zone ticket. 8. The Passenger selects zone 4. The Distributor displays $2.25. The Passenger inserts a $1 bill and a quarter, and selects zone 2. **The Distributor returns the $1 bill and the quarter and display $1.25.**
Exit condition	The Passenger has two 2-zone tickets, one 1-zone ticket, and one 4-zone ticket.

5. System Testing – Global Properties

- Some system properties are inherently global
  - Performance, latency, robustness, ...
  - Early and incremental testing is still necessary, but provide only estimates
- A major focus of system testing
  - The only opportunity to verify global properties against actual system specifications
  - Especially to find unanticipated effects, e.g., an unexpected performance bottleneck

Context-Dependent Properties

- Beyond system-global: Some properties depend on the system context and use
  - Example: Performance properties depend on environment and configuration
  - Example: Privacy depends both on system and how it is used
    - Medical records system must protect against unauthorized use, and authorization must be provided only as needed
  - Example: Security depends on threat profiles
    - And threats change!

Establishing an Operational Envelope

- When a property (e.g., performance or real-time response) is parameterized by use ...
  - requests per second, size of database, ...
- Extensive stress testing is required
  - varying parameters within the envelope, near the bounds, and beyond
- Goal: A well-understood model of how the property varies with the parameter
  - How sensitive is the property to the parameter?
  - Where is the "edge of the envelope"?
  - What can we expect when the envelope is exceeded?

Stress Testing

- Often requires extensive simulation of the execution environment
  - With systematic variation: What happens when we push the parameters? What if the number of users or requests is 10 times more, or 1000 times more?
- Often requires more resources (human and machine) than typical test cases
  - Separate from regular feature tests
  - Run less often, with more manual control
  - Diagnose deviations from expectation
    - Which may include difficult debugging of latent faults!

Estimating Reliability

- Measuring quality, not searching for faults
  - Fundamentally different goal than systematic testing
- Quantitative reliability goals are statistical
  - Failure probability
  - Failure intensity
  - Availability
  - Mean time to failure
  - ...
- Requires valid statistical samples from *operational profile*
  - Fundamentally different from systematic testing

Estimating Reliability (cont.)

- We need a valid *operational profile* (model)
  - Sometimes from an older version of the system
  - Sometimes from operational environment (e.g., for an embedded controller)
  - *Sensitivity testing* reveals which parameters are most important, and which can be rough guesses
- And a clear, precise definition of what is being measured
  - Failure rate? Per session, per hour, per operation?
- And many, many random samples
  - Especially for high reliability measures

Usability

- A usable product
  - is quickly learned
  - allows users to work efficiently
  - is pleasant to use
- Objective criteria
  - Time and number of operations to perform a task
  - Frequency of user error
    - blame user errors on the product!
- Plus overall, subjective satisfaction

Usability (cont.)

Verifying Usability

- Usability rests ultimately on testing with real users — validation, not verification
  - Preferably in the usability lab, by usability experts
- But we can *factor* usability testing for process visibility — validation *and verification* throughout the project
  - Validation establishes criteria to be verified by testing, analysis, and inspection

Usability (cont.)

Factoring Usability Testing

Validation (usability lab)

- Usability testing establishes usability check-lists
  – Guidelines applicable across a product line or domain
- Early usability testing evaluates "cardboard prototype" or mock-up
  – Produces interface design

Verification (developers, testers)

- Inspection applies usability check-lists to specification and design
- Behavior objectively verified (e.g., tested) against interface design

Usability (cont.)

Varieties of Usability Test

- **Exploratory** testing
  - Investigate mental model of users
  - Performed early to guide interface design

- **Comparison** testing
  - Evaluate options (specific interface design choices)
  - Observe (and measure) interactions with alternative interaction patterns

- **Usability** validation testing
  - Assess overall usability (quantitative and qualitative)
  - Includes measurement: error rate, time to complete

Usability (cont.)

Typical Usability Test Protocol

- Select *representative sample* of user groups
  - Typically 3-5 users from each of 1-4 groups
  - Questionnaires verify group membership

- Ask users to perform a representative sequence of tasks

- Observe without interference (no helping!)
  - The hardest thing for developers is to *not help*. Professional usability testers use one-way mirrors.

- Measure (clicks, eye movement, time, ...) and follow up with questionnaire