Software Testing

Chapter 8

Software Testing
in the textbook

- Introduction (Verification and Validation)
- 8.1 Development testing
- 8.2 Test-driven development
- 8.3 Release testing
- 8.4 User testing

Verification and Validation

- Verification:
  - The software should conform to its specification (the functional and non-functional requirements).

    "Are we building the product right".
- Validation:
  - The software should do what the customer really requires.

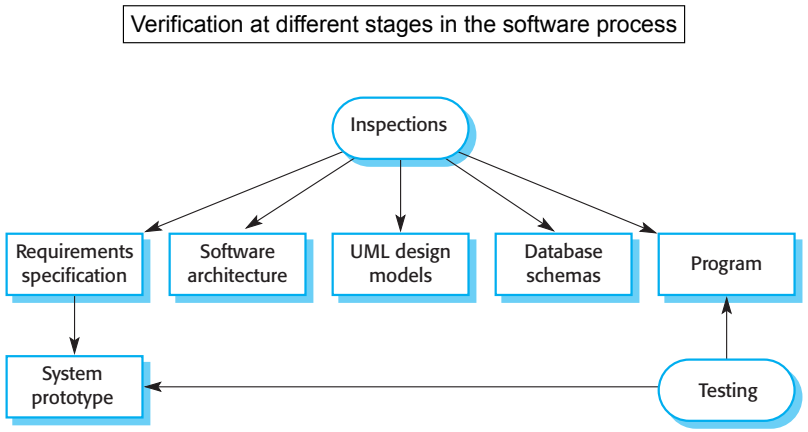
    "Are we building the right product".

Requirements don't always reflect real wishes and needs of customers and users

Verification Techniques

- Static verification
  - Inspections, reviews
  - Analyze static system representations to discover problems
  - Applies to: specifications, design models, source code, test plans
- Dynamic verification
  - Testing
  - The system is executed with simulated test data
  - Check results for errors, anomalies, and data regarding non-functional requirements.

Verification Techniques



Testing Concepts

- Failure
  - Deviation between the specification and the actual behavior of the system.
- Fault (aka “bug” or “defect”)
  - A design or coding mistake that may cause abnormal behavior (with respect to specifications)
- Test case
  - set of inputs and expected results that exercises a system (or part) with the purpose of detecting faults
- Testing
  - the systematic attempt to find faults in a planned way in the implemented software.

Test cases

Test cases should contain the following:

- Name
  - Explains what is being tested
- Input
  - Set of input data and/or commands and/or actions
- Expected results
  - Output or state or behavior that is correct for the given input.

Black-box and White-box testing

Different kinds of test cases:

- Black-box tests
  - focus on input/output behavior of the software
  - are not based on how the software is structured or implemented.
- White-box tests:
  - focus on the internal structure of the software
  - an internal perspective of the system is used to design test cases.
  - goal: test all parts of code in the software

Test stubs and drivers

How to test units/components in isolation:

- test driver
  - code that simulates the part of the system that calls the component under test.
  - often provides the input for a given test case
  - this code is in a function that is executed during the test
- test stub
  - code that simulates a component that is called by the tested component
  - must support the called component's interface, and return a value of the appropriate type.

Testing Activities/Methods

For each kind of testing activity we consider:

- Who performs the tests?
  - Developers, independent testing team, users, customers
- What are the constraints of the tests?
  - test a certain part of the system?
  - test the system at a certain point in the process?
- How are the test-cases developed?
  - where does the data come from?
  - what data to use?
  - what functionality is tested?

Software testing activities:
Who performs the test?

- 8.1 Development testing:
  - developers test the system during development
- 8.3 Release testing:
  - a separate testing team tests a complete version of the system before it is released to users.
- 8.4 User testing:
  - Customers and users (or potential users) of a system test the system in their own environment.

8.1 Development testing
Which parts are tested?

- Unit testing
  - individual program units (i.e. classes) are tested
- Component testing
  - system components (composed of individual units) are tested to make sure the contained units interact correctly.
- System testing
  - the system components are integrated and the system is tested as a whole.

Unit testing

- Unit testing:
  - individual program units are tested in isolation
  - focus is on testing functionality of the units
- Goal: complete test coverage of a class:
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states
- Why focus on such small units?
  - reduces complexity of overall test activities
  - makes it easier to pinpoint faults
  - different objects can be tested concurrently

Unit testing: How are test cases developed?

- Partition (or equivalence) testing:
  - identify groups of inputs that have common characteristics and should be processed the same way by the system
- Guideline based testing
  - use guidelines that reflect the kinds of errors programmers often make
- Path testing
  - exercise all possible paths through the code at least once
- State-based testing
  - define sequences of events to force all possible transitions.

Partition testing

- Divide the set of all possible input data of a software unit into partitions
  - program should behave similarly for all data in a given partition
  - Determine partitions from specifications
- Design one test case for each partition, using test input from the given partition.
  - If the partition is a range, also test boundary values.
- Enables good test coverage with fewer test cases.

Partition testing: example

Function returning the number of days in a month:

```
int getNumDaysInMonth(int month, int year);
```

Partition	month value	year value
Month with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Month with 30 days, leap year	6 (June)	1904
Month with 28 or 29 days, non-leap year	2 (February)	1901
Month with 28 or 29 days, leap year	2 (February)	1904

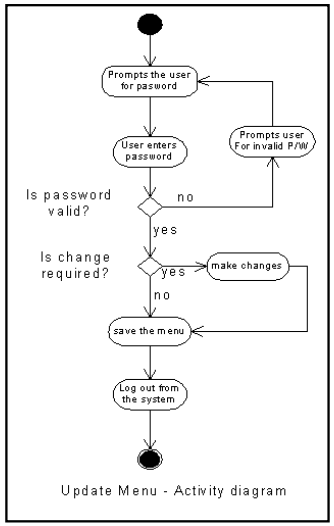
Guideline-based testing

- Choose test cases based on previous experience of common programming errors
- For example:
  - Choose inputs that force the system to generate all error messages
  - Repeat the same input or series of inputs numerous times
  - Try to force invalid outputs to be generated
  - Force computation results to be too large or too small.
  - Test sequences/lists using
    - one element
    - zero elements
    - different sizes in different tests

Path testing

- Exercise all possible paths through the code at least once
  - a white-box testing technique
  - convert code to control-flow (or UML activity) diagram
  - choose input data so that each path through diagram is executed

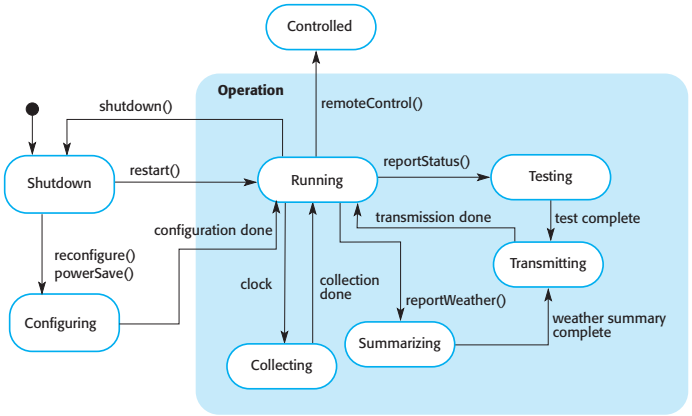
Example: Make sure that at least one test case forces each oval to execute:
- one with valid password
- one with invalid password
- one that requires a change
- one that doesn't require a change



State-based testing

- Define sequences of events to force all possible transitions in a UML state diagram
  - Identify sequences of state transitions to be tested
  - Write test cases that generates the event sequences to cause these transitions.
  - Verify that the program ends up in the expected state.
- sequences from the diagram on the next slide:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing -> Transmitting -> Running
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running

State-based testing: example (weather station)

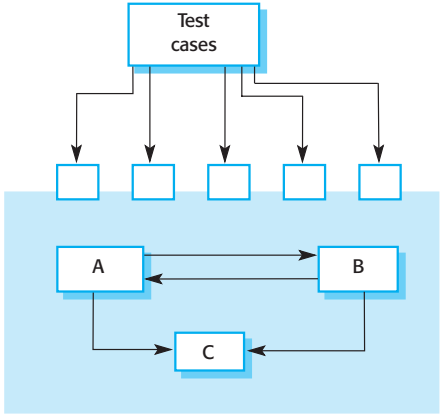


Shutdown -> Running-> Shutdown is tested with a call to restart() followed by a call to shutdown(), then check the state

Component testing

- Component testing
  - System components (composed of individual units) are tested to make sure the units interact correctly.
  - The functionality of these objects is accessed through the defined component interface.
- Component testing is demonstrating that the component interface behaves according to its specification.
  - Assuming the subcomponents (objects) have already been unit-tested

Interface (component) testing



Small empty boxes represent the interface

System testing

- System testing
  - the components in a system are integrated and the system is tested as a whole.
- Checks that:
  - components are compatible,
  - components interact correctly
  - components transfer the right data at the right time across their interfaces.
- Tests the interactions between components.

8.3 Release testing

- Release Testing
  - testing a particular release of a system that is intended for use outside of the development team.
- Similar to system testing, but
  - Tested by a team other than developers.
  - Focus is on demonstrating system meets requirements.
- Primary goal: convince the supplier of the system that it is good enough for use.
- Black-box testing process where tests are derived from the system specification.

System and Release testing: How are test cases developed?

- Use case-based testing:
  - use the use-cases developed during requirements engineering to develop test cases.
- Scenario testing
  - use scenarios (user stories) developed during requirements engineering to develop test cases.
- Requirements-based testing
  - examine each requirement in the SRS and develop one or more tests for it.

Use case-based testing: example

Use case name	PurchaseTicket
Entry condition	The Passenger is standing in front of ticket Distributor. The Passenger has sufficient money to purchase ticket.
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 tickets. 7. The Passenger picks up the change and the ticket.
Exit condition	The Passenger has the selected ticket.

Developing the test from the use case:

- The PurchaseTicket use case describes the normal interaction between the Passenger actor and the Distributor.
- 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.

Purchase Ticket use case test case

Test case name	PurchaseTicket_Common Case
Entry condition	The Passenger is standing in front of ticket Distributor. The Passenger has two $5 bills and three dimes.
Flow of events	1.The Passenger presses in succession the zone buttons 2, 4, 1, and 2. 2.The Distributor displays 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 1-4 using his second $5 bill. 6.The Passenger repeats steps 1-3 using four quarters and three dimes. The Distributor issues a 2-zone ticket and returns a nickel. 7.The Passenger selects zone 1 and inserts a dollar bill. The Distributor issues a 1-zone ticket and returns a quarter. 8.The Passenger selects zone 4 and inserts two $1 bills and a quarter. The Distributor issues a 4-zone ticket. 9.The Passenger selects zone 4. The Distributor displays $2.25. The Passenger inserts a $1 bill and a nickel, and selects zone 2. The Distributor returns the $1 bill and the nickel and displays $1.25.
Exit condition	The Passenger has three 2-zone tickets, one 1-zone ticket, and one 4-zone ticket.

Scenario testing

- A scenario is a story that describes one way in which the system might be used
  - Longer than an “interaction”
- To use a scenario for release testing:
  - tester assumes role of user, acting out scenario
  - may make deliberate mistakes (as part of the scenario)
  - takes note of problems (slow response, errors, etc.)
- Tests several requirements and interactions at once, in combination.
- See example in book: Figure 8.10, section 8.3.2

Requirements-based testing

- Example requirements from MHC-PMS system:

1. If a patient is recorded as being allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
2. The system shall allow the prescriber to override an allergy warning by providing a reason why this has been ignored and the prescription will succeed. If no reason is provided, the prescription will fail.

Requirements-based testing

Some tests developed to test the previous requirement:

1. Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is **not** issued by the system, and that the prescription succeeds.
2. Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system. Accept the warning and make sure the prescription fails.
3. Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system. Provide a reason overruling the warning and make sure the prescription succeeds.

8.4 User testing

- User testing:
  - Customers and users (or potential users) of a system test the system in their own environment.
- Essential even when comprehensive system and release testing have been carried out.
  - Influences from the user’s working environment have a major effect on the reliability, performance, usability and robustness of a system.
  - These cannot be replicated in a testing environment.

Types of user testing

- Alpha testing
  - Users of the software work with the development team to test the software at the developer's site.
  - generic or custom software
- Beta testing
  - A release of the software is made available to users to allow them to experiment and to report problems
  - generic or custom software
- Acceptance testing
  - Customers test a system to decide whether or not it is ready to be accepted from the system developers.
  - acceptance implies payment is due, may require negotiation.
  - custom software.