CS 410/510 - Software Engineering

Software Testing

Reference: Sommerville, Software Engineering, 10 ed., Chapter 8

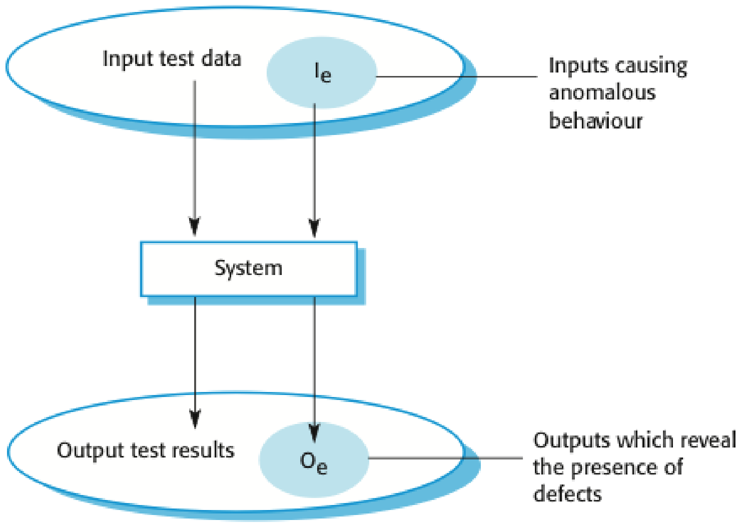
The big picture

Testing is intended to show that a **program does what it is intended to do** and to **discover program defects** before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies or information about the program's non-functional attributes. **Testing can reveal the presence of errors, but NOT their absence.** Testing is part of a more general verification and validation process, which also includes static validation techniques.

Goals of software testing:

- To **demonstrate** to the developer and the customer that the **software meets its requirements**.
    - Leads to **validation testing**: you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
    - A successful test shows that the system operates as intended.
- To **discover** situations in which the behavior of the software is **incorrect, undesirable or does not conform to its specification**.
    - Leads to **defect testing**: the test cases are designed to expose defects; the test cases can be deliberately obscure and need not reflect how the system is normally used.
    - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Testing can be viewed as an **input-output process**:


Input test data
Ie
Inputs causing anomalous behaviour
System
Output test results
Oe
Outputs which reveal the presence of defects

Verification and validation

Testing is part of a broader process of software **verification and validation** (V & V).

- **Verification: Are we building the product right?**
  The software should conform to its specification.
- **Validation: Are we building the right product?**
  The software should do what the user really requires.

The **goal of V & V** is to establish confidence that the system is **good enough for its intended use**, which depends on:

- **Software purpose**: the level of confidence depends on how critical the software is to an organization.
- **User expectations**: users may have low expectations of certain kinds of software.
- **Marketing environment**: getting a product to market early may be more important than finding defects in the program.

Inspections and testing

Software inspections involve people examining the source representation with the aim of discovering anomalies and defects. Inspections not require execution of a system so may be used before implementation. They may be applied to any representation of the system (requirements, design,configuration data, test data, etc.). They have been shown to be an effective technique for discovering program errors.

Advantages of inspections include:

- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with **interactions between errors**.
- **Incomplete versions** of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider **broader quality attributes** of a program, such as compliance with standards, portability and maintainability.

Inspections and testing are complementary and not opposing verification techniques. Both should be used during the V & V process. Inspections can check conformance with a specification but not conformance with the customer's real requirements. Inspections cannot check non-functional characteristics such as performance, usability, etc.

Typically, a commercial software system has to go through **three stages of testing**:

- **Development testing**: the system is tested during development to discover bugs and defects.
- **Release testing**: a separate testing team test a complete version of the system before it is released to users.
- **User testing**: users or potential users of a system test the system in their own environment.

Development testing

Development testing includes all testing activities that are carried out by the team developing the system:

- **Unit testing**: individual program units or object classes are tested; should focus on testing the functionality of objects or methods.
- **Component testing**: several individual units are integrated to create composite components; should focus on testing component interfaces.
- **System testing**: some or all of the components in a system are integrated and the system is tested as a whole; should focus on testing component interactions.

Unit testing

Unit testing is the process of **testing individual components in isolation**. It is a defect testing process. Units may be:

- **Individual functions** or methods within an object;
- **Object classes** with several attributes and methods;
- **Composite components** with defined interfaces used to access their functionality.

When **testing object classes**, tests should be designed to provide coverage of all of the features of the object:

- Test **all operations** associated with the object;
- Set and check the **value of all attributes** associated with the object;
- Put the object into **all possible states**, i.e. simulate all events that cause a state change.

Whenever possible, **unit testing should be automated** so that tests are run and checked without manual intervention. In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests. Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests. An automated test has three parts:

- A **setup** part, where you initialize the system with the test case, namely the inputs and expected outputs.
- A **call** part, where you call the object or method to be tested.
- An **assertion** part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.

The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do. If there are defects in the component, these should be revealed by test cases. This leads to **two types of unit test cases**:

- The first of these should reflect **normal operation** of a program and should show that the component works as expected.
- The other kind of test case should be based on testing experience of where common problems arise. It should use **abnormal inputs** to check that these are properly processed and do not crash the component.

Component testing

Software components are often composite components that are **made up of several interacting objects**. You access the functionality of these objects through the **defined component interface**. Testing composite components should therefore focus on showing that the component interface behaves according to its specification. Objectives are to detect faults due to interface errors or invalid assumptions about interfaces. Interface types include:

- **Parameter interfaces**: data passed from one method or procedure to another.
- **Shared memory interfaces**: block of memory is shared between procedures or functions.
- **Procedural interfaces**: sub-system encapsulates a set of procedures to be called by other sub-systems.
- **Message passing interfaces**: sub-systems request services from other sub-systems.

Interface errors:

- **Interface misuse**: a calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- **Interface misunderstanding**: a calling component embeds assumptions about the behavior of the called component which are incorrect.
- **Timing errors**: the called and the calling component operate at different speeds and out-of-date information is accessed.

General guidelines for interface testing:

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

System testing

System testing during development involves **integrating components** to create a version of the system and then **testing the integrated system**. The focus in system testing is **testing the interactions between components**. System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces. System testing tests the **emergent behavior** of a system.

During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested. Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.

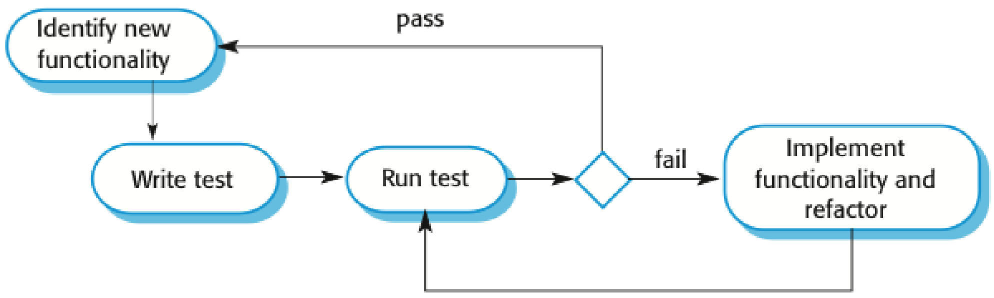
The **use cases** developed to identify system interactions can be used as a **basis for system testing**. Each use case usually involves several system components so testing the use case forces these interactions to occur. The **sequence diagrams** associated with the use case **document the components and their interactions** that are being tested.

Test-driven development

Test-driven development (TDD) is an approach to program development in which you **inter-leave testing and code development**. **Tests are written before code** and 'passing' the tests is the critical driver of development. This is a differentiating feature of TDD versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code. **The code is developed incrementally, along with a test for that increment**. You don't move on to the next increment until the code that you have developed passes its test. TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

TDD example - a string calculator.

The **goal of TDD** isn't to ensure we write tests by writing them first, but to **produce working software that achieves a targeted set of requirements using simple, maintainable solutions**. To achieve this goal, TDD provides strategies for **keeping code working, simple, relevant, and free of duplication**.


Identify new functionality
pass
Write test
Run test
fail
Implement functionality and refactor

TDD process includes the following activities:

1. Start by **identifying the increment of functionality** that is required. This should normally be small and implementable in a few lines of code.
2. **Write a test** for this functionality and implement this as an automated test.
3. **Run the test**, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
4. **Implement the functionality and re-run the test.**
5. Once all tests run successfully, you move on to implementing the next chunk of functionality.

Benefits of test-driven development:

- **Code coverage**: every code segment that you write has at least one associated test so all code written has at least one test.
- **Regression testing**: a regression test suite is developed incrementally as a program is developed.
- **Simplified debugging**: when a test fails, it should be obvious where the problem lies; the newly written code needs to be checked and modified.
- **System documentation**: the tests themselves are a form of documentation that describe what the code should be doing.

Regression testing is testing the system to check that **changes have not 'broken' previously working code**. In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program. Tests must run 'successfully' before the change is committed.

Release testing

Release testing is the process of **testing a particular release** of a system that is **intended for use outside of the development team**. The primary goal of the release testing process is to **convince the customer of the system that it is good enough for use**. Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use. Release testing is usually a black-box testing process where **tests are only derived from the system specification**.

Release testing is a form of system testing. Important differences:

- A separate team that has not been involved in the system development, should be responsible for release testing.
- System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

Requirements-based testing involves examining each requirement and developing a test or tests for it. It is validation rather than defect testing: you are trying to demonstrate that the system has properly implemented its requirements.

Scenario testing is an approach to release testing where you devise typical scenarios of use and use these to develop test cases for the system. Scenarios should be realistic and real system users should be able to relate to them. If you have used scenarios as part of the requirements engineering process, then you may be able to reuse these as testing scenarios.

Part of release testing may involve testing the **emergent properties** of a system, such as performance and reliability. Tests should reflect the profile of use of the system. Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable. Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.

User testing

User or customer testing is a stage in the testing process in which **users or customers provide input and advice on system testing**. User testing is essential, even when comprehensive system and release testing have been carried out. Types of user testing include:

- **Alpha testing**: users of the software work with the development team to test the software at the developer's site.
- **Beta testing**: a release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- **Acceptance testing**: customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system. Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made. Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.