Software Testing

Some of the slides taken from:

1) Software Engineering, Ian Sommerville, 9th Edition
2) Prof. Richard McKenna’s lecture on Test Driven Development at SBU

Topics covered

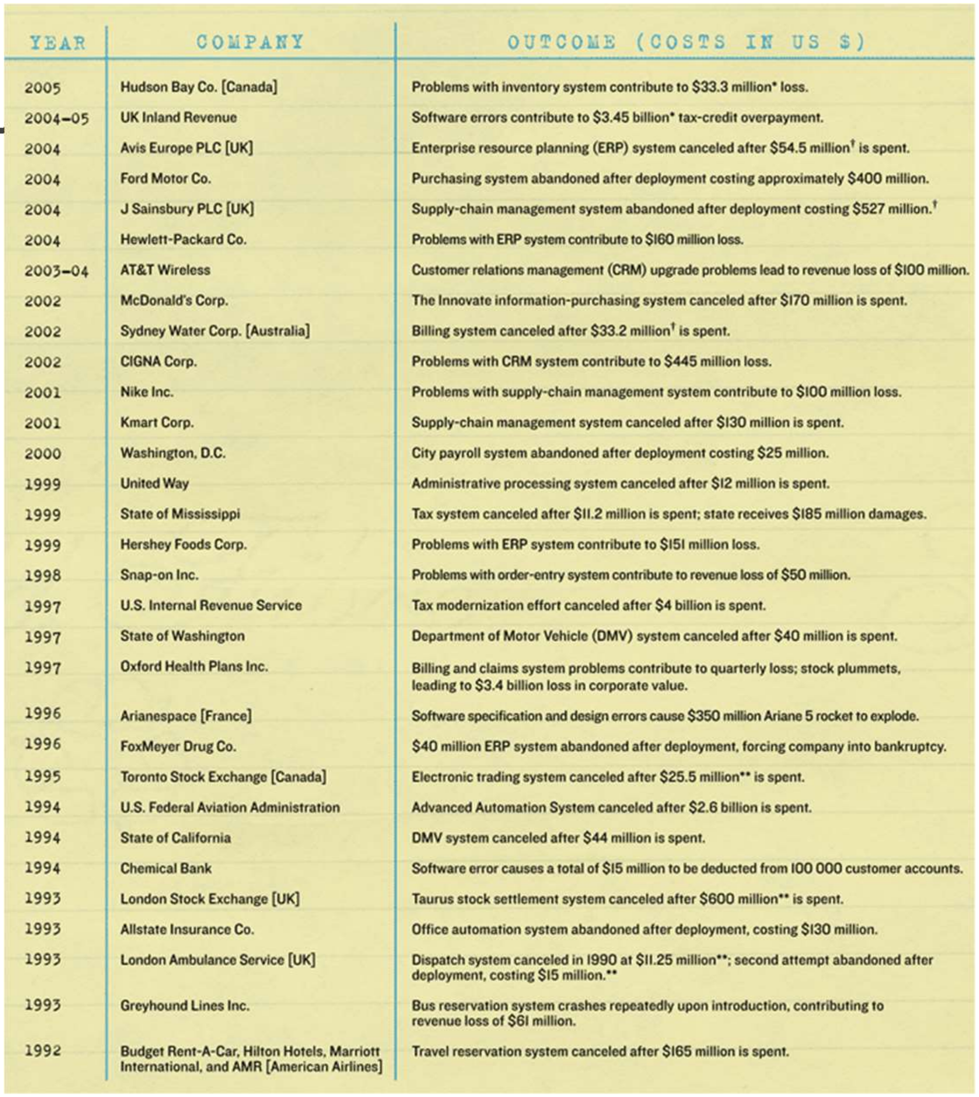
- ✧ Why do software projects fail?
- ✧ Development testing
- ✧ Test-driven development
- ✧ Release testing
- ✧ User testing

Reading Assignment

Why Software Fails:

https://spectrum.ieee.org/computing/software/why-software-fails

Software Hall of Shame ->



YEAR	COMPANY	OUTCOME (COSTS IN US $)
2005	Hudson Bay Co. [Canada]	Problems with inventory system contribute to $33.3 million* loss.
2004–05	UK Inland Revenue	Software errors contribute to $3.45 billion* tax-credit overpayment.
2004	Avis Europe PLC [UK]	Enterprise resource planning (ERP) system canceled after $54.5 million† is spent.
2004	Ford Motor Co.	Purchasing system abandoned after deployment costing approximately $400 million.
2004	J Sainsbury PLC [UK]	Supply-chain management system abandoned after deployment costing $527 million.†
2004	Hewlett-Packard Co.	Problems with ERP system contribute to $160 million loss.
2003–04	AT&T Wireless	Customer relations management (CRM) upgrade problems lead to revenue loss of $100 million.
2002	McDonald's Corp.	The Innovate information-purchasing system canceled after $170 million is spent.
2002	Sydney Water Corp. [Australia]	Billing system canceled after $33.2 million† is spent.
2002	CIGNA Corp.	Problems with CRM system contribute to $445 million loss.
2001	Nike Inc.	Problems with supply-chain management system contribute to $100 million loss.
2001	Kmart Corp.	Supply-chain management system canceled after $130 million is spent.
2000	Washington, D.C.	City payroll system abandoned after deployment costing $25 million.
1999	United Way	Administrative processing system canceled after $12 million is spent.
1999	State of Mississippi	Tax system canceled after $11.2 million is spent; state receives $185 million damages.
1999	Hershey Foods Corp.	Problems with ERP system contribute to $151 million loss.
1998	Snap-on Inc.	Problems with order-entry system contribute to revenue loss of $50 million.
1997	U.S. Internal Revenue Service	Tax modernization effort canceled after $4 billion is spent.
1997	State of Washington	Department of Motor Vehicle (DMV) system canceled after $40 million is spent.
1997	Oxford Health Plans Inc.	Billing and claims system problems contribute to quarterly loss; stock plummets, leading to $3.4 billion loss in corporate value.
1996	Arianespace [France]	Software specification and design errors cause $350 million Ariane 5 rocket to explode.
1996	FoxMeyer Drug Co.	$40 million ERP system abandoned after deployment, forcing company into bankruptcy.
1995	Toronto Stock Exchange [Canada]	Electronic trading system canceled after $25.5 million** is spent.
1994	U.S. Federal Aviation Administration	Advanced Automation System canceled after $2.6 billion is spent.
1994	State of California	DMV system canceled after $44 million is spent.
1994	Chemical Bank	Software error causes a total of $15 million to be deducted from 100 000 customer accounts.
1993	London Stock Exchange [UK]	Taurus stock settlement system canceled after $600 million** is spent.
1993	Allstate Insurance Co.	Office automation system abandoned after deployment, costing $130 million.
1993	London Ambulance Service [UK]	Dispatch system canceled in 1990 at $11.25 million**; second attempt abandoned after deployment, costing $15 million.**
1993	Greyhound Lines Inc.	Bus reservation system crashes repeatedly upon introduction, contributing to revenue loss of $61 million.
1992	Budget Rent-A-Car, Hilton Hotels, Marriott International, and AMR [American Airlines]	Travel reservation system canceled after $165 million is spent.

WHY DO PROJECTS FAIL SO OFTEN?

Among the most common factors:

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project’s status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Use of immature technology
- Inability to handle the project’s complexity
- Sloppy development practices
- Poor project management
- Stakeholder politics
- Commercial pressures

Man-hours

- Labor is sometimes measured in man-hours, man-months, or man-years.
  - Example: Doom3 took 5 years and more than 100 man-years of labor to develop
    - Company Spokesman: "It will be ready when it's done"
  - Why not double the size of the team and halve the *lead time* (concept date to release date)?

Man-hours: The Mythical Man-Month

- Assume that a software program might take one expert programmer a year to develop = 12 man-months
- Market pressures might be such that we want to get the program finished in a month, rather than a year
- 1 programmer * 12 months = 12 programmers * 1 month?
  - When you throw additional programmers at a project that is late, you are likely to make it *more late!*
  - ***Remove promised-but-not-yet-completed features, rather than multiplying workers bees.***
  - ***Also, at least one team member must have detailed knowledge of the entire system (all the modules).***

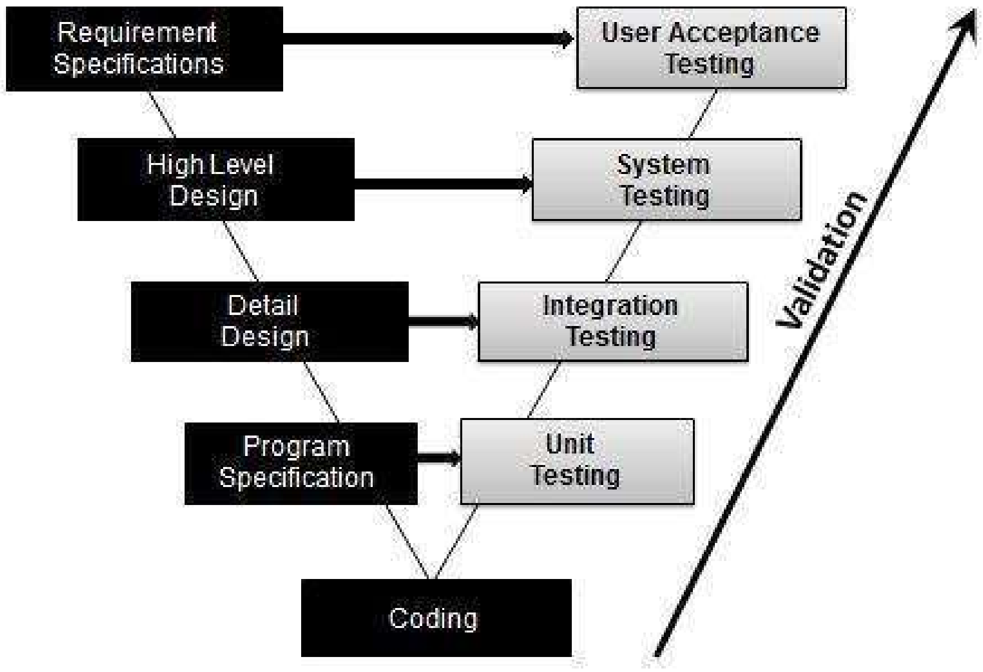
Program testing

- 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.
- Can reveal the presence of errors NOT their absence.
- Testing is part of a more general verification and validation process, which also includes static validation techniques.

Testing process goals

✧ Validation testing

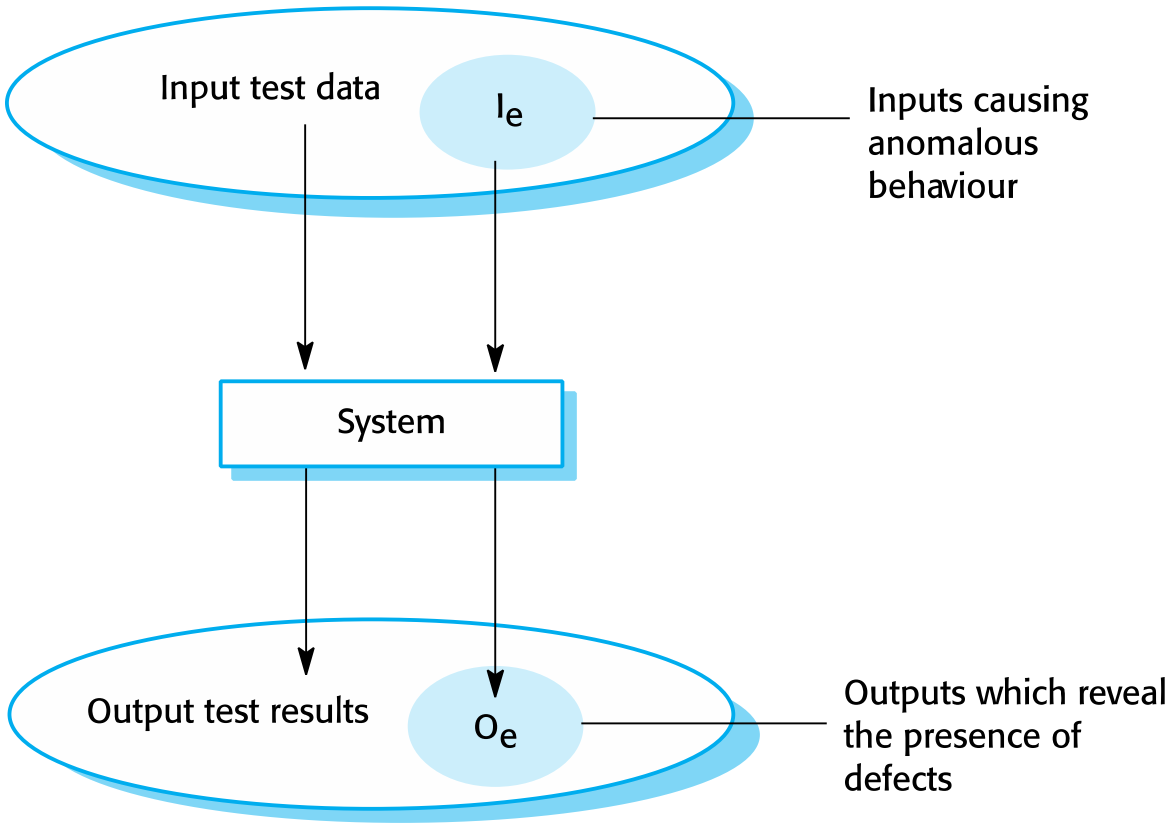
- To demonstrate to the developer and the system customer that the software meets its requirements.
- A successful test shows that the system operates as intended.
- You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.



Testing process goals

- ✧ Defect testing
  - A Software DEFECT / BUG is a condition in a software product which does not meet a software requirement (as stated in the requirement specifications) or end-user expectation (which may not be specified but is reasonable).
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification.
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.
  - The test cases are designed to expose defects.
  - The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

An input-output model of program testing


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

Verification vs validation

- ✧ Verification:
  "Are we building the product right".
- ✧ The software should conform to its specification.
- ✧ Whether the system is well-engineered, error-free.
- ✧ Validation:
  "Are we building the right product".
- ✧ The software should do what the user really requires.

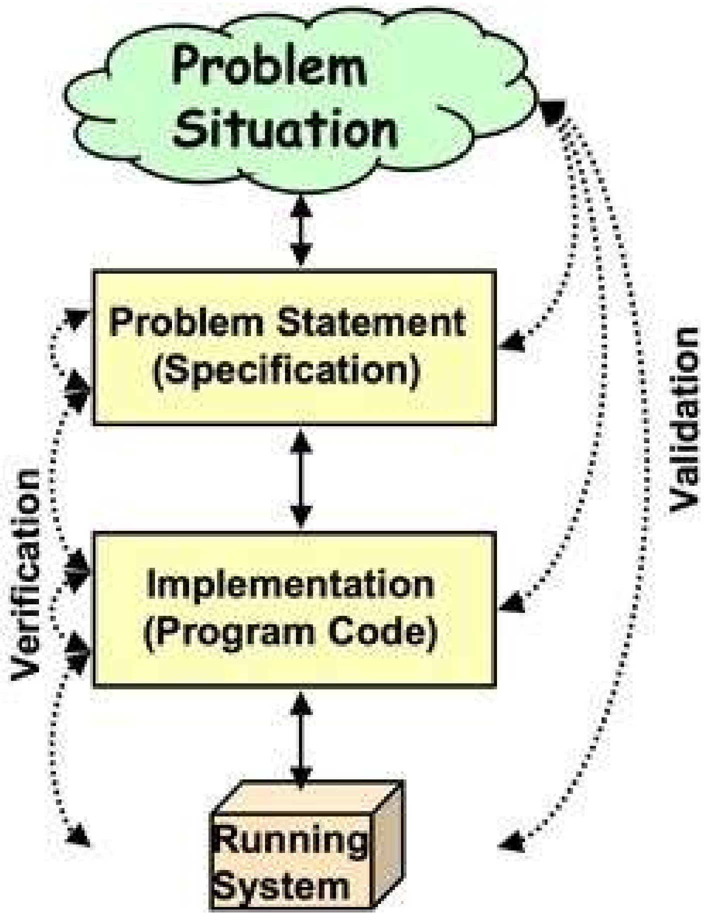
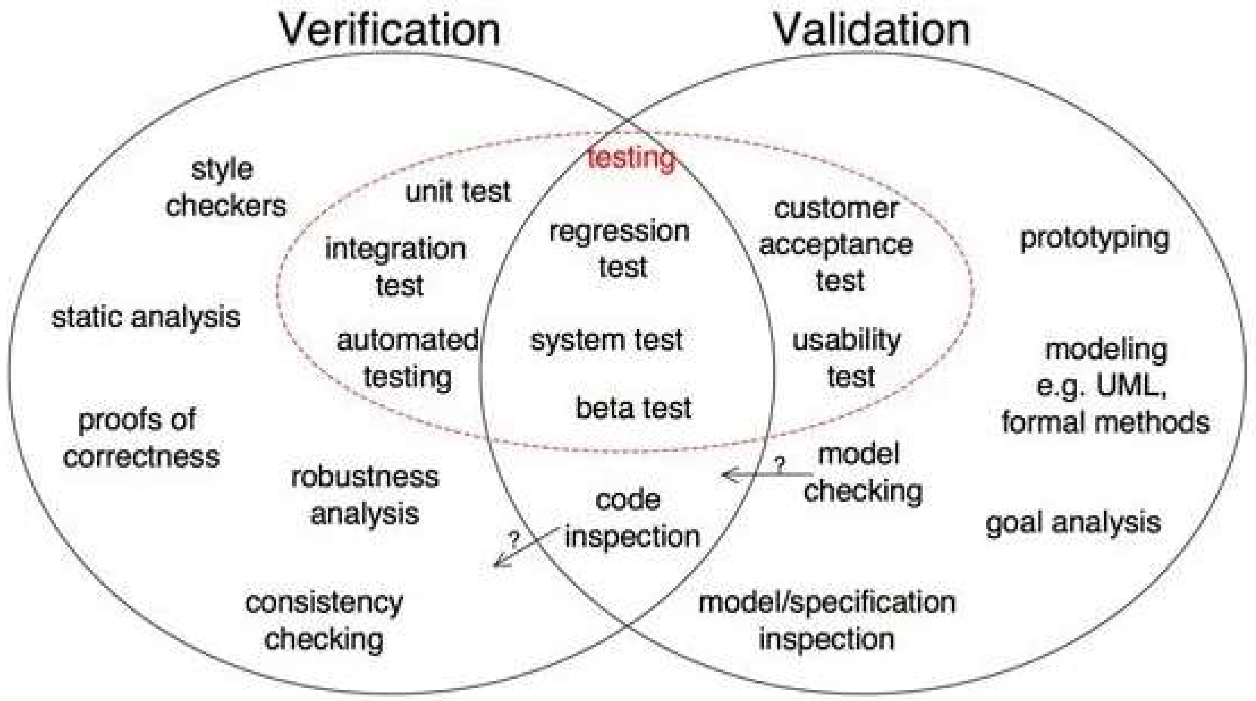

Problem Situation
Problem Statement (Specification)
Implementation (Program Code)
Running System
Verification
Validation

Figure source: https://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/

Verification vs validation



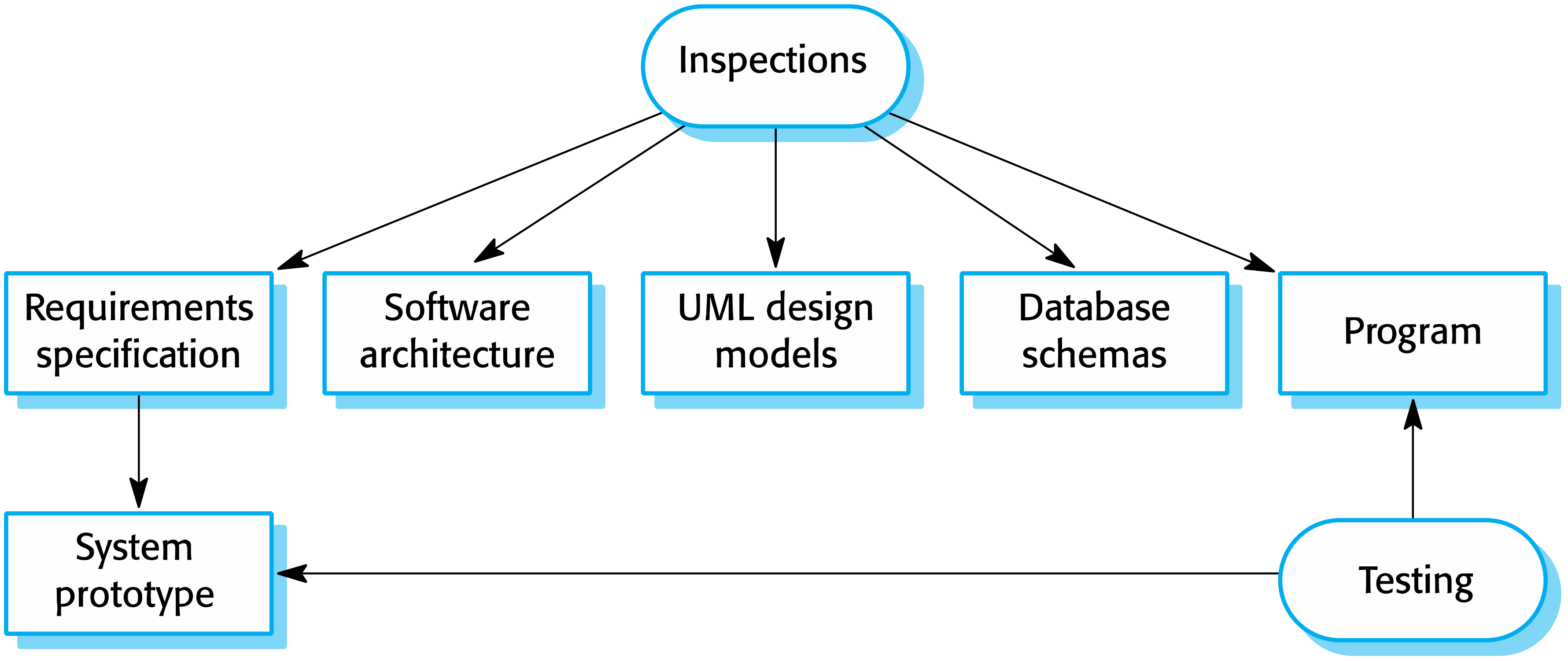
A range of V&V techniques. Note that "modeling" and "model checking" refer to building and analyzing abstracted models of software behaviour, a very different kind of beast from scientific models used in the computational sciences

Figure source: https://www.easterbrook.ca/steve/2010/11/the-difference-between-verification-and-validation/

Inspections and testing

- ✧ Software inspections is concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis.
- ✧ Software testing is concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed.

Inspections and testing



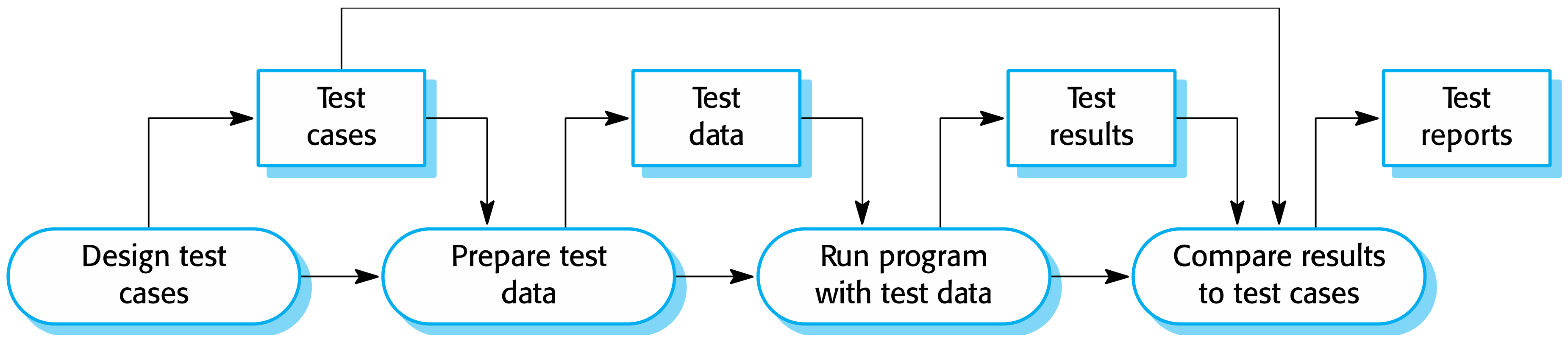
Software inspections

- ✧ These 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.

Inspections and testing

- ✧ 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.

A model of the software testing process



Stages of testing

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

General testing guidelines

- ✧ Choose inputs that force the system to generate all error messages.
- ✧ Design inputs that cause input buffers to overflow.
- ✧ Repeat the same input or series of inputs numerous times.
- ✧ Force invalid outputs to be generated.
- ✧ Force computation results to be too large or too small.

Development testing

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.
  - Unit testing should focus on testing the functionality of objects or methods.
- Component testing
  - Several individual units are integrated to create composite components.
  - Component testing 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.
  - System testing 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.

Object class testing

- ✧ Complete test coverage of a class involves:
  - Testing all operations associated with an object.
  - Setting and interrogating all object attributes.
  - Exercising the object in all possible states.
- ✧ Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

Automated testing

- ✧ 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 on the success of otherwise of the tests.

Automated test components

- 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.

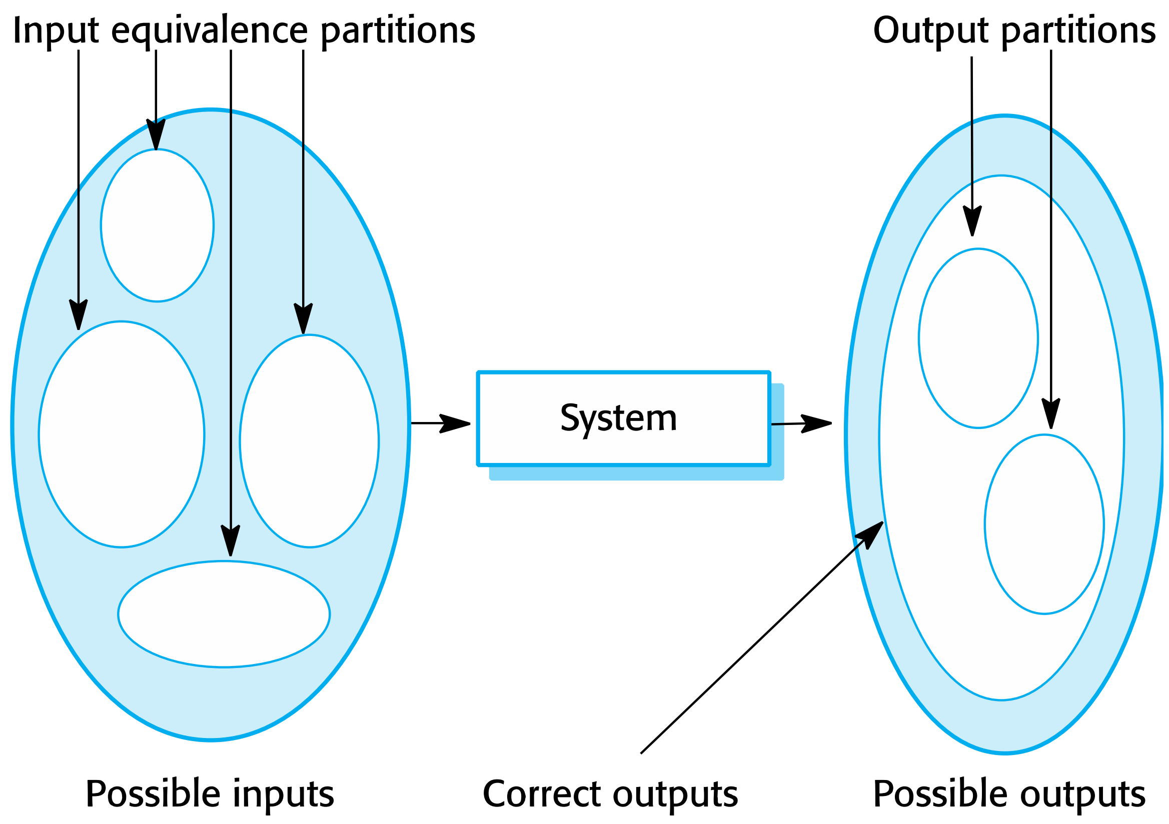
Boundary Conditions

- ✧ A boundary condition is an input that is “one away” from producing a different behavior in the program code.
- ✧ Such checks catch 2 common types of errors:
  - ▪ Logical errors, in which a path to handle a special case presented by a boundary condition is omitted.
  - ▪ Failure to check for conditionals that may cause the underlying language or hardware system to raise an exception (ex: arithmetic overflow).

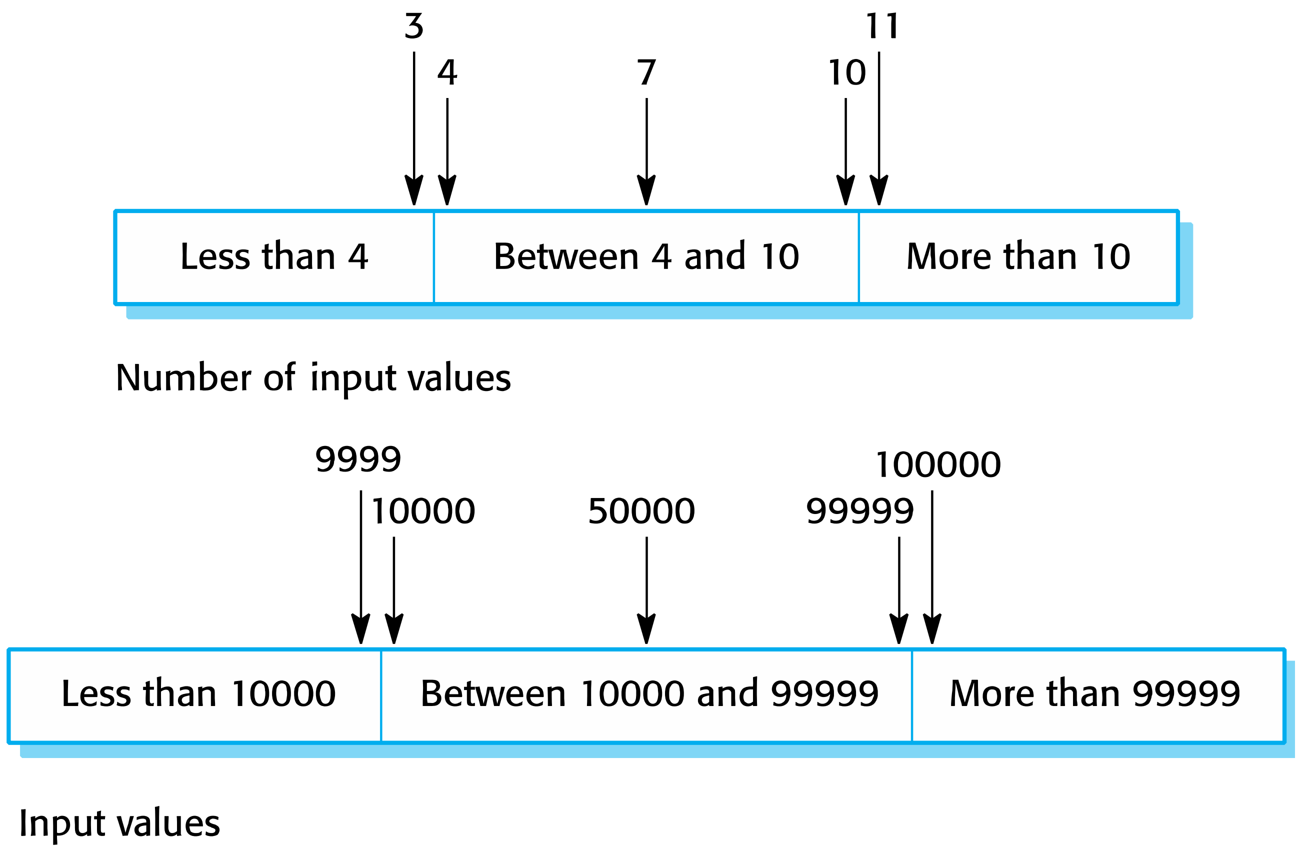
Partition testing

- ✧ Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
- ✧ Input data and output results often fall into different classes where all members of a class are related.
- ✧ Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- ✧ Test cases should be chosen from each partition.

Equivalence partitioning



Equivalence partitions



Testing paths through specification

- Path-completeness:
  - Test cases are generated to exercise each path through a program.
  - May be insufficient to catch all errors.
  - Can be used effectively only for a program fragment that contains a reasonable number of paths to test.
  - Examine the method specifications (preconditions) & all paths through method to generate unique test cases for testing.

```
/* REQUIRES: x >= 0 && y >= 10 */

public static int calc(int x, int y) { … }
```

- Translate paths to test cases:

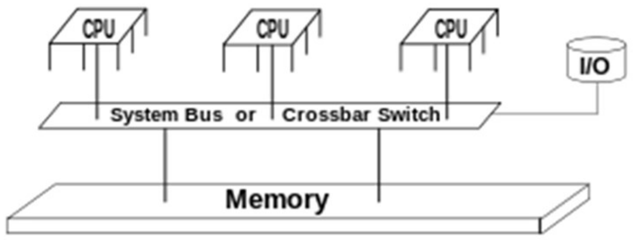
```
x =  0, y = 10 (x == 0 && y == 10)
x =  5, y = 10 (x >  0 && y == 10)
x =  0, y = 15 (x == 0 && y >  10)
x =  5, y = 15 (x >  0 && y >  10)
x = -1, y = 10 (x <  0 && y == 10)
x = -1, y = 15 (x <  0 && y >  10)
x = -1, y =  9 (x <  9 && y <  10)
x =  0, y =  9 (x == 0 && y <  10)
x =  1, y =  9 (x >  0 && y <  10)
```

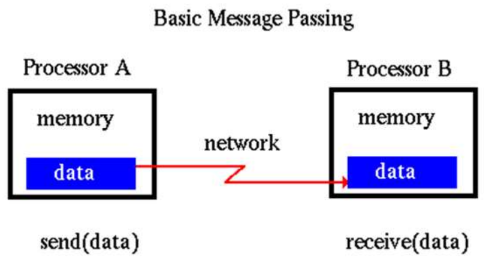
Component testing

- ✧ Software components are often composite components that are made up of several interacting objects.
  - ▪ For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- ✧ 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.
  - ▪ You can assume that unit tests on the individual objects within the component have been completed.

Interface testing

- ✧ Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- ✧ Interface types
  - **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 testing guidelines

- ✧ 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.

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.

System and component testing

- ✧ 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.
- ✧ In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

Black-box testing

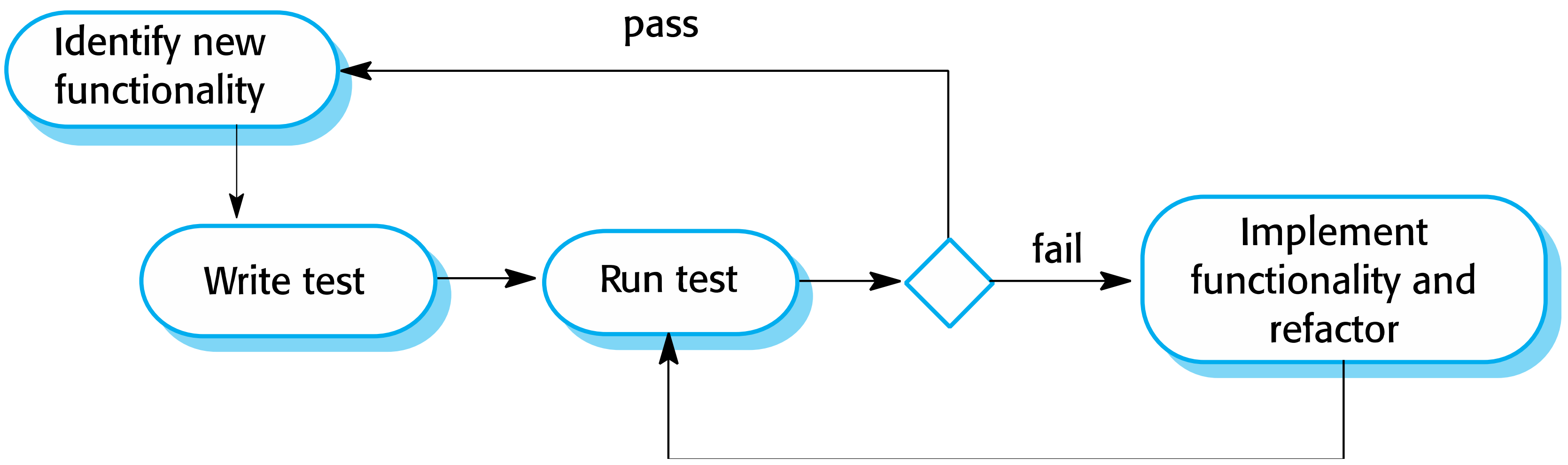
- It is the best place to start when attempting to test a program thoroughly
- Test cases based on program's specification, not on its implementation
- Test cases are not affected by:
  - Invalid assumptions made by the programmer
  - Implementation changes
    - Use same test cases even after program structures has changed
- Test cases can be generated by an "independent" agent, unfamiliar with the implementation.
- Test cases should cover all paths (not all cases) through the specification, including exceptions.

Test-driven development

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.
- You develop code 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.

Test-driven development


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

TDD process activities

- ✧ Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- ✧ Write a test for this functionality and implement this as an automated test.
- ✧ 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.
- ✧ Implement the functionality and re-run the test.
- ✧ 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

- ✧ 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

- ✧ 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 supplier 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 and system testing

- ✧ 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).

Performance testing

- ✧ 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 behaviour.

User testing

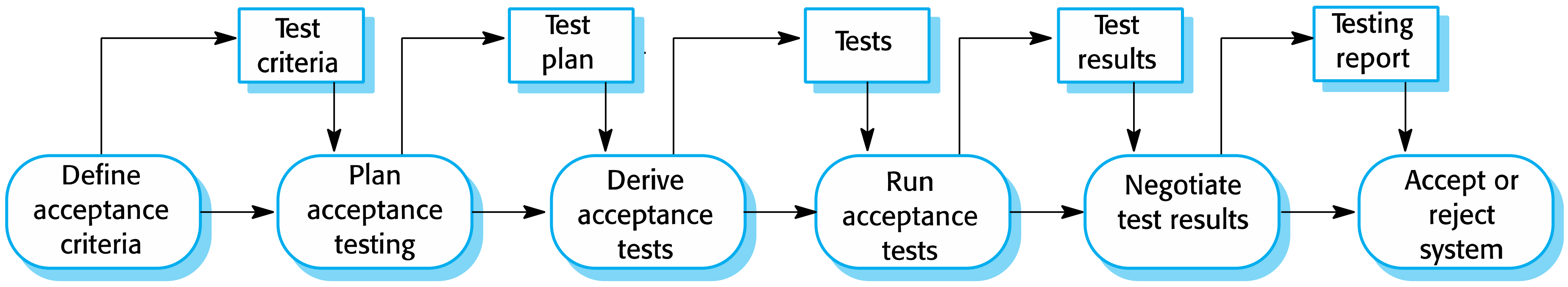
User testing

- ✧ 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.
  - The reason for this is that 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.
- ✧ 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. Primarily for custom systems.

The acceptance testing process



Stages in the acceptance testing process

- ✧ Define acceptance criteria
- ✧ Plan acceptance testing
- ✧ Derive acceptance tests
- ✧ Run acceptance tests
- ✧ Negotiate test results
- ✧ Reject/accept system

JUnit

JUnit

- Unit-test framework for Java programs.
  - open source software
  - hosted on SourceForge: http://junit.sourceforge.net/javadoc
    - Moved to http://junit.org (for JUnit 4 and later)
  - not in the standard JDK:

```
import junit.framework.*;
                    //for JUnit 3.8 and earlier
import org.junit.*; //for JUnit 4 and later
```

- Associate a Test class with each unit
  - one or more classes

JUnit

- ✧ The test class has a set of test methods

```
public void testX()
```

where **`x`** is the method to be tested

- ✧ The test methods use “assertions” to perform the tests, ex:

```
Assert.assertEquals(x,y)
Assert.assertTrue(c)
Assert.assertSame(obj1, obj2)
```

Building unit tests with JUnit

- ✧Initialize any instance variables necessary for testing in the test object
- ✧Define tests for emptiness, equality, boundary conditions, ...
- ✧Define test suites, if necessary, to group tests.
- ✧Use Assert methods to perform tests

JUnit 3.8 vs. 4

- JUnit 4: all test methods are annotated with @Test.
  - Unlike JUnit3 tests, you do not need to prefix the method name with "test“.
- JUnit 4 does not have the test classes extend junit.framework.TestCase (directly or indirectly).
  - Usually, tests with JUnit4 do not need to extend anything (which is good, since Java does not support multiple inheritance).

JUNIT tutorials with example

- ✧ Test-Driven Development with Junit:
- ✧ https://www.youtube.com/watch?v=2Ekty7t621k
- ✧ Junit NetBeans Example:
- ✧ https://examples.javacodegeeks.com/core-java/junit/junit-netbeans-example/

Apache JMeter