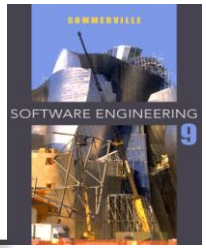


Chapter 8 – Software Testing

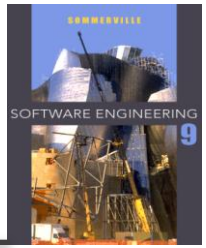
Ian Sommerville,

Software Engineering, 9th Edition

Pearson Education, Addison-Wesley

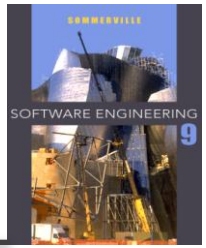
Note: These are a modified version of Ch 8 slides available from the author's site http://www.cs.st-andrews.ac.uk/~ifs/Books/SE9/

Topics covered



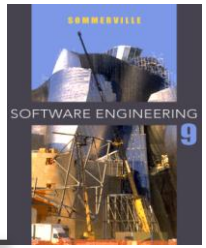
- ✧ Development testing
- ✧ Test-driven development
- ✧ Release testing
- ✧ User testing

Program testing



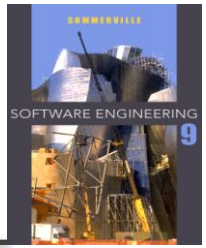
- Testing shows that a program does what it is intended to do and to discover program defects before it is put into use. It is a dynamic validation and verification (V&V) technique.
- To test, you execute a program using artificial data
- Check the results of the test run for errors, anomalies or information 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 V&V techniques

Program testing goals



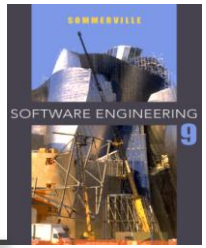
- To demonstrate that *software meets its requirements*
  - For custom software, at least one test for every requirement in the requirements document
  - For generic software products, have tests for all of the system features AND feature combinations
- To discover situations in which the behavior of the software is *incorrect or undesirable*
  - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption

Validation and defect testing



- The first goal 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
- The second goal leads to defect testing
  - The test cases are designed to expose defects. Tests can be obscure and address unusual use of software.

Testing process goals



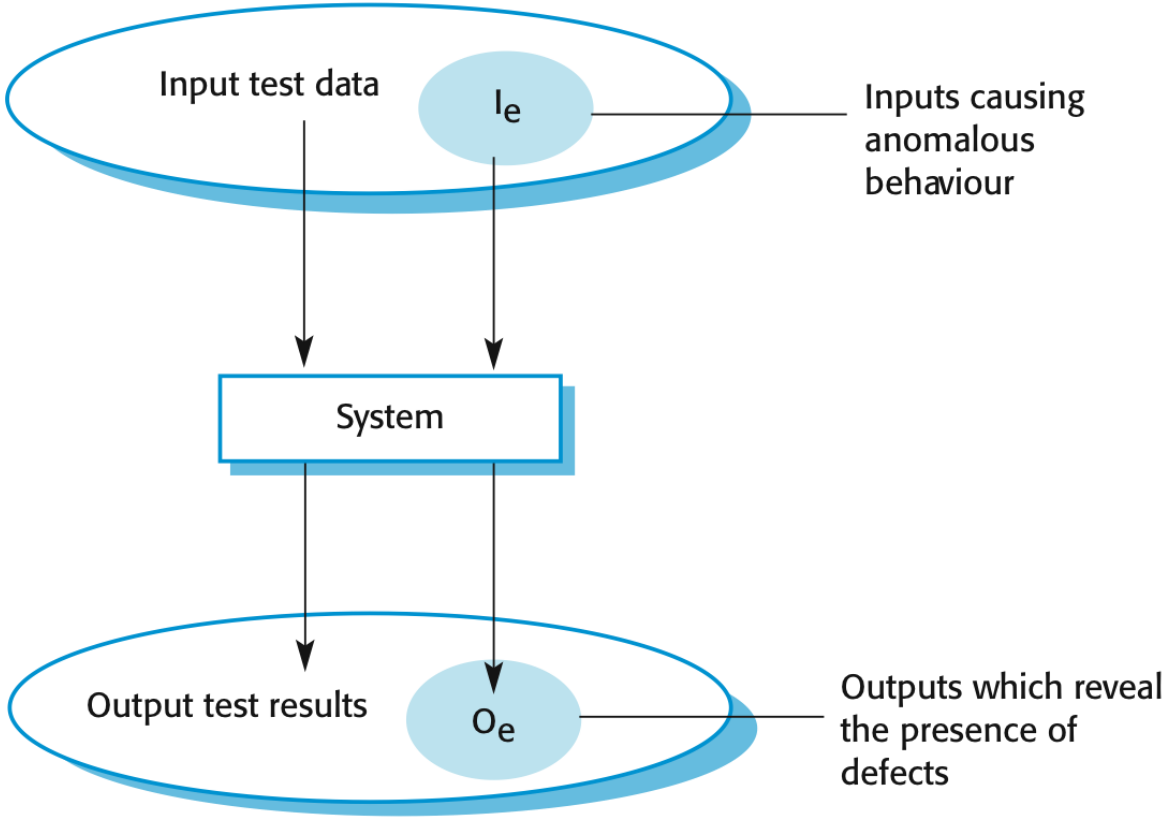
✧ Validation testing

- 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

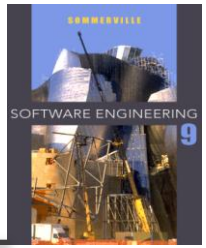
✧ Defect testing

- To discover faults or defects in the software where its behavior is incorrect or doesn’t follow its specification
- A successful test is a test that makes the system perform incorrectly

An input-output model of program testing



Verification vs validation



✧ Verification:

"*Are we building the product right?*"

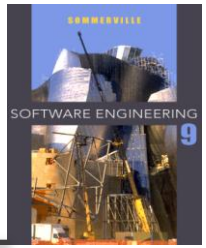
- The software should conform to its specification

✧ Validation:

"*Are we building the right product?*"

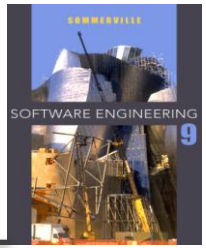
- The software should do what the user really requires

V & V confidence



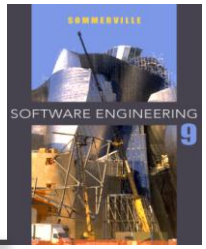
- Aim of V & V is to establish confidence that the system is '**fit for purpose**'
- Depends on:
  - Software purpose
    - The level of confidence depends on how critical the software is to an organization
  - User expectations
    - Users may have low expectations of certain kinds of software
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program

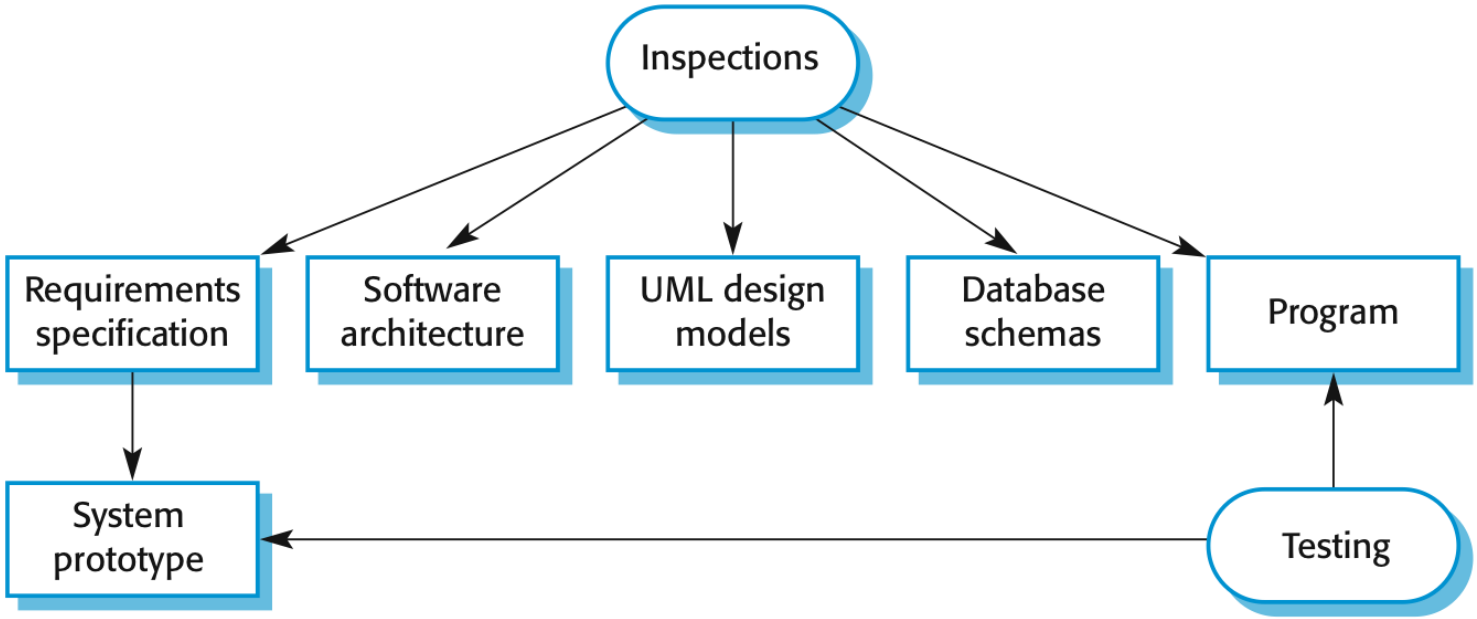
Inspections and testing



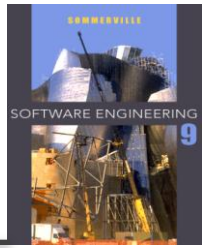
- Software inspections: Analyze the static system representation to discover problems (static verification)
  - May be supplemented by tool-based document and code analysis (discussed in Chapter 15).
- Software testing: Exercise and observe product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed.

Inspections and testing



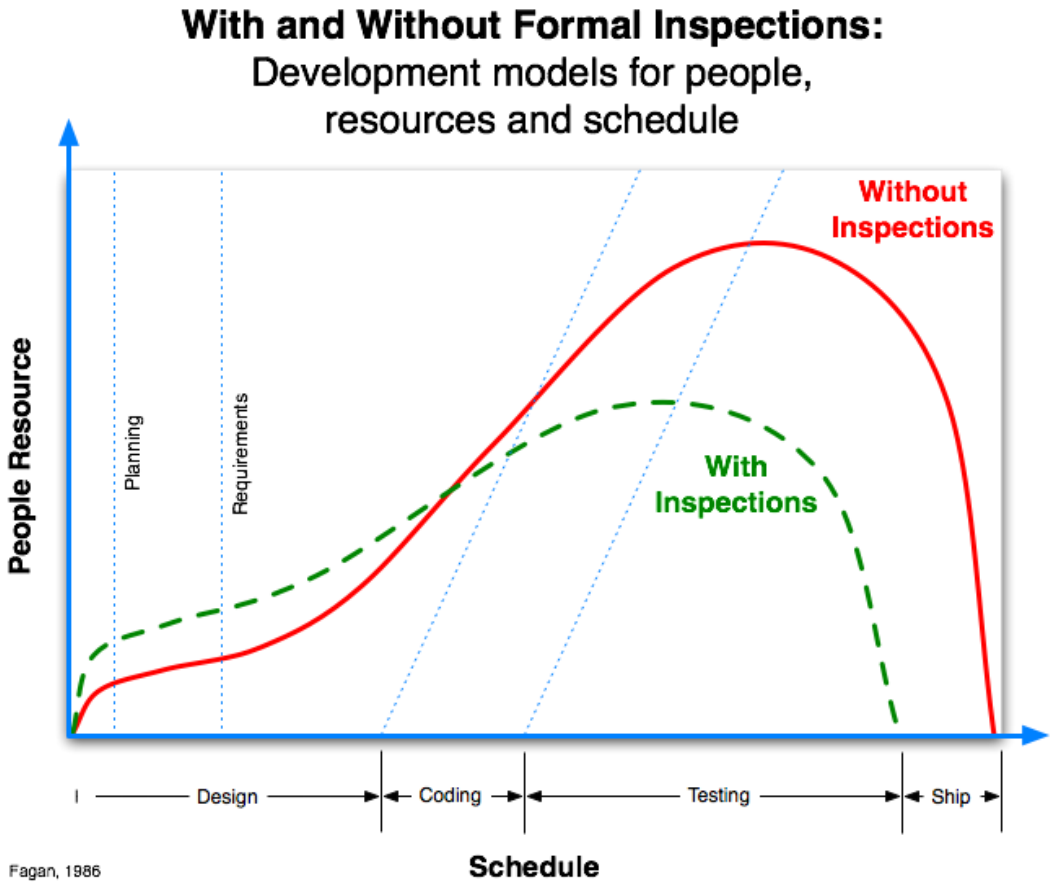


Software inspections



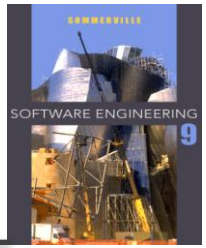
- People examine the source code to discover anomalies and defects
- Inspections do not require execution of a system.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.)
- They have been shown to be an effective technique for discovering program errors

Software Inspections



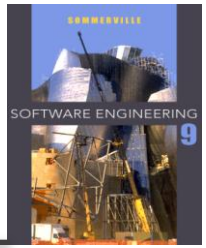
http://weblog.bosslogic.com/2007/07/formal-inspection-an-introduction/

Advantages of inspections



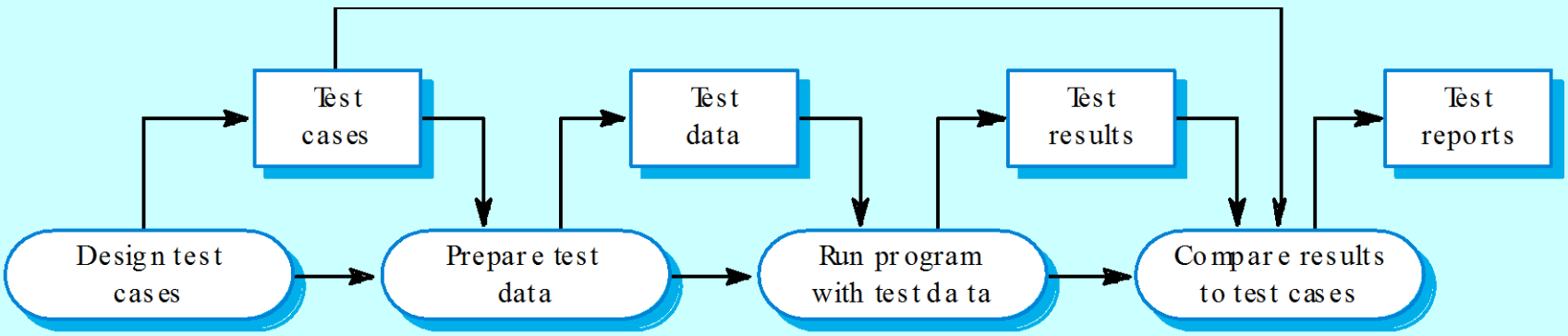
- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.
- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized 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

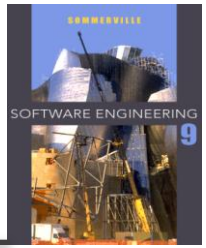


- Inspections and testing are complementary and not opposing verification techniques
- Both should be used during the V & V process
- Inspections can check conformance with a specification but not conformance with the customer's real requirements
- Inspections cannot check non-functional characteristics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

The software testing process

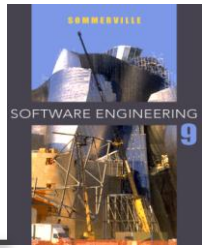


Stages of testing



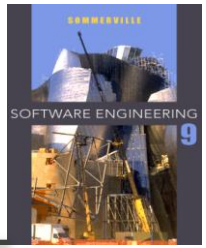
- Development testing - the system is tested during development
- Release testing - a separate testing team tests a complete version of the system before it is released
- User testing - users or potential users of a system test the system in their own environment

Development testing



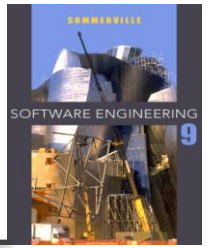
- Development testing includes all testing activities that are carried out by the developers
  - Unit testing: focuses on testing the functionality of objects or methods
  - Component testing: create components from object combinations. Focuses on testing component interfaces.
  - System testing: the components in a system are integrated and the system is tested as a whole. Focuses on testing component interactions.

Unit testing



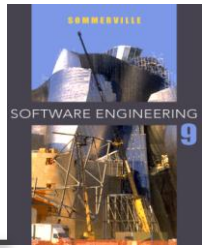
- Unit testing is the process of testing individual components in isolation
- The purpose is to discover defects
- Units may be:
  - Individual functions or methods within a class
  - Classes with several attributes and methods
  - Composite components with defined interfaces used to access their functionality

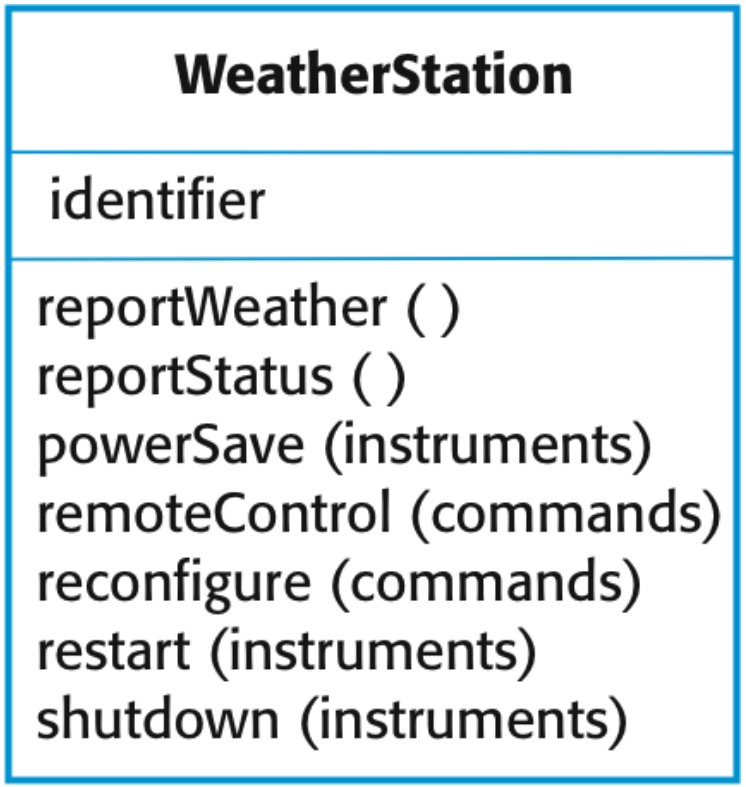
Object class testing



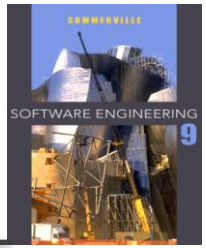
- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states
- Inheritance makes it more difficult to design object class tests

The weather station object interface



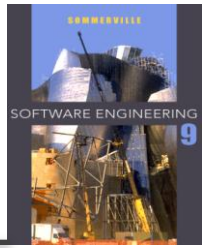


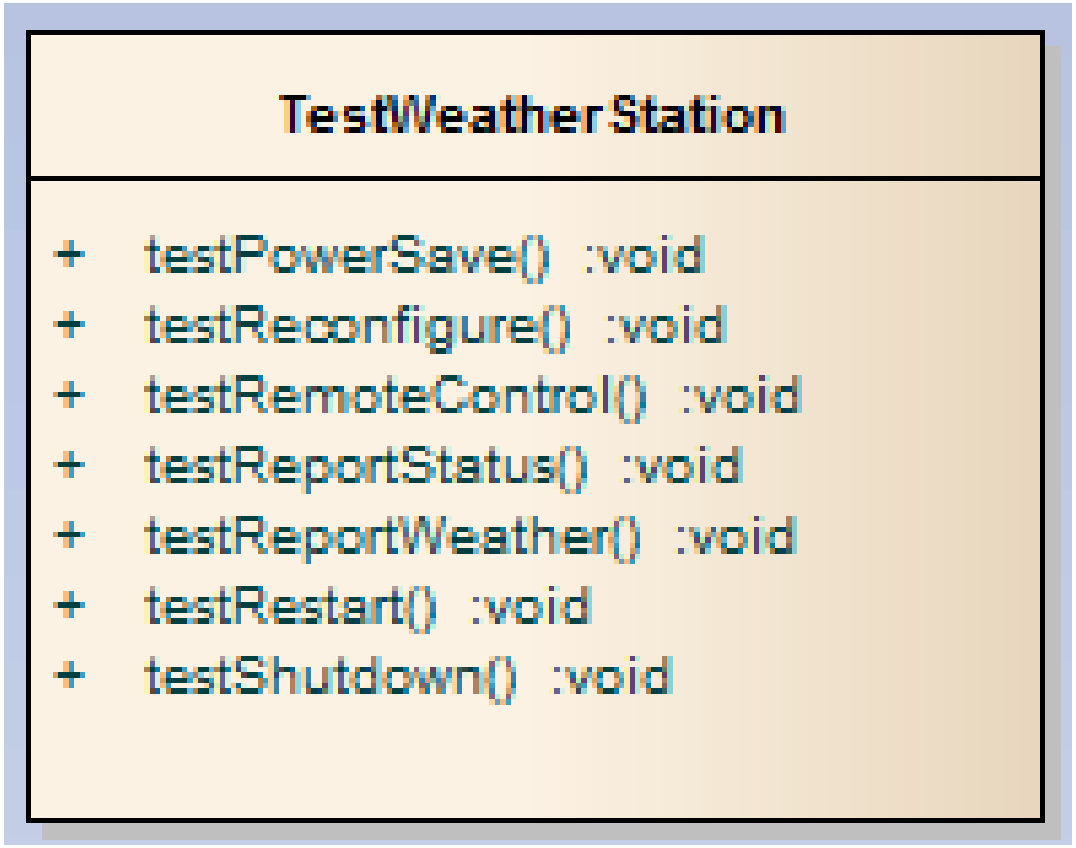
Weather station testing



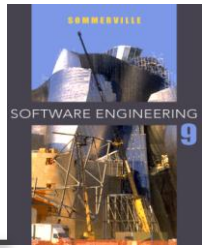
- Need to define test cases for all operations.
- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- For example:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing -> Transmitting -> Running
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running

The Weather Station Testing Object



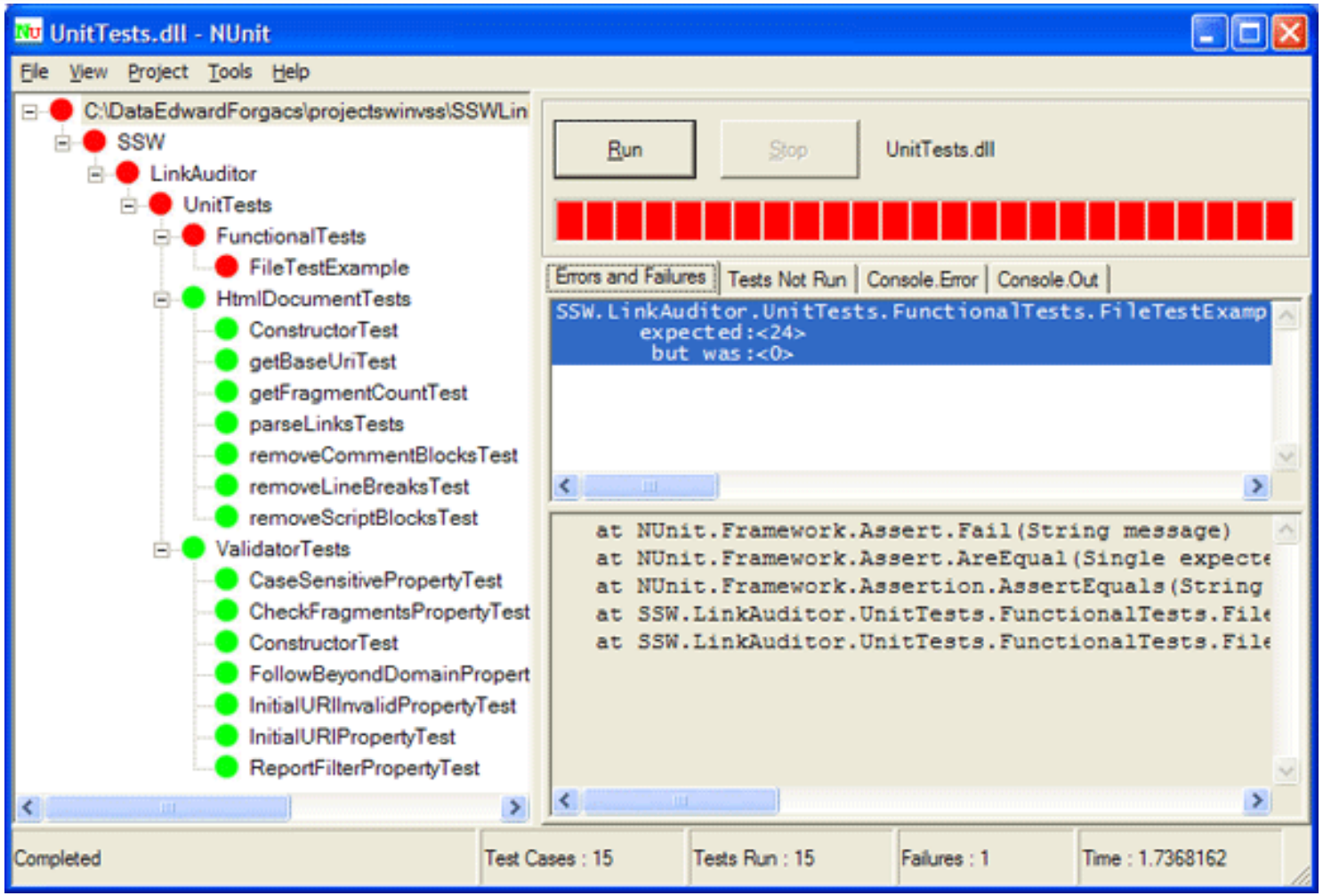


Automated testing

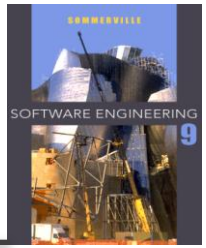


- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention
- In automated unit testing, the testers make use of a test automation framework (such as JUnit) to write and run your program tests
- Unit testing frameworks can run all the implemented tests and report on the success of the tests

Automated Testing

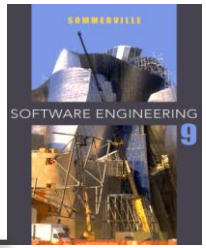


Unit test effectiveness



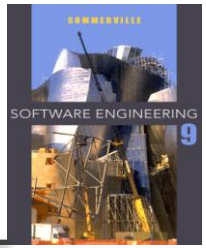
- The test cases should show that component does what it is supposed to do
- Should reveal defects in the component, if there are any
- This leads to 2 types of unit test cases:
  - Tests that reflect normal operation of a program to show that the meets expectations.
  - Tests based on testing experience of where common problems arise. Should use abnormal inputs, check that these are properly processed, and not crash the component.

Testing strategies



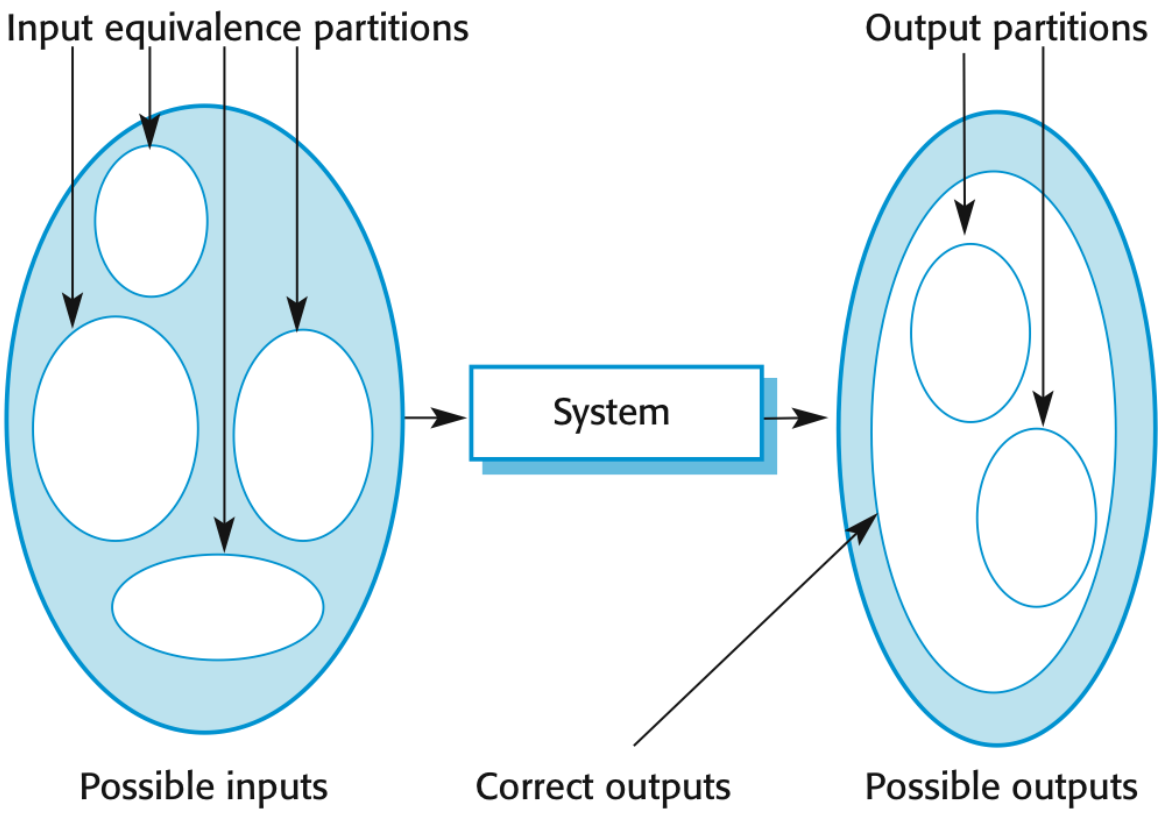
- Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way
  - You should choose tests from within each of these groups
- Guideline-based testing, where you use testing guidelines to choose test cases
  - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Partition testing

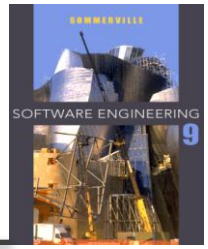


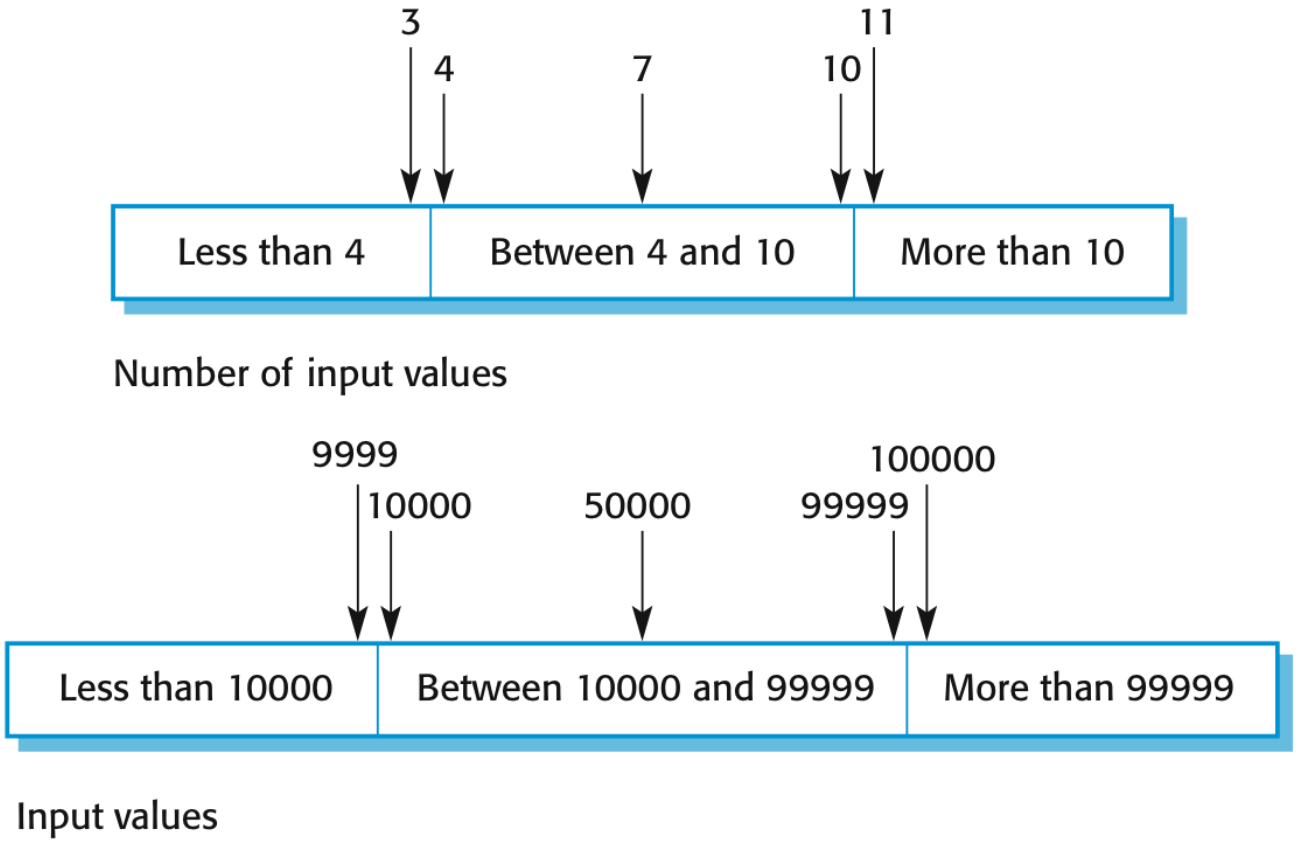
- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition

Equivalence partitioning

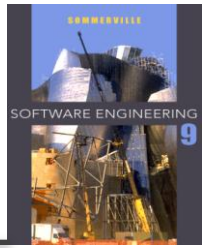


Equivalence partitions





Search routine specification



procedure Search (Key : ELEM ; T: SEQ of ELEM;
Found : **in out** BOOLEAN; L: **in out** ELEM_INDEX) ;

Pre-condition

-- the sequence has at least one element
T'FIRST <= T'LAST

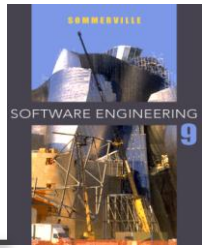
Post-condition

-- the element is found and is referenced by L
(Found and T (L) = Key)

or

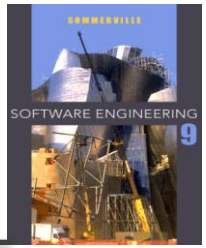
-- the element is not in the array
(**not** Found **and**
not (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key))

Search routine - input partitions



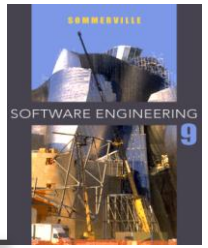
- ✧ Inputs which conform to the pre-conditions.
- ✧ Inputs where a pre-condition does not hold.
- ✧ Inputs where the key element is a member of the array.
- ✧ Inputs where the key element is not a member of the array.

Testing guidelines (sequences, lists, arrays)



- ✧ Test software with sequences which have only a single value
- ✧ Use sequences of different sizes in different tests
- ✧ Derive tests so that the first, middle and last elements of the sequence are accessed

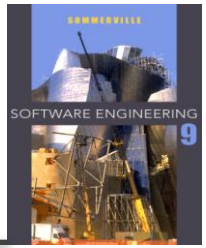
Search routine - input partitions



Sequence	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

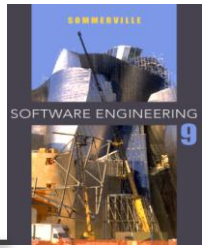
Input sequence (T)	**Key** (Key)	**Output** (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

General testing guidelines



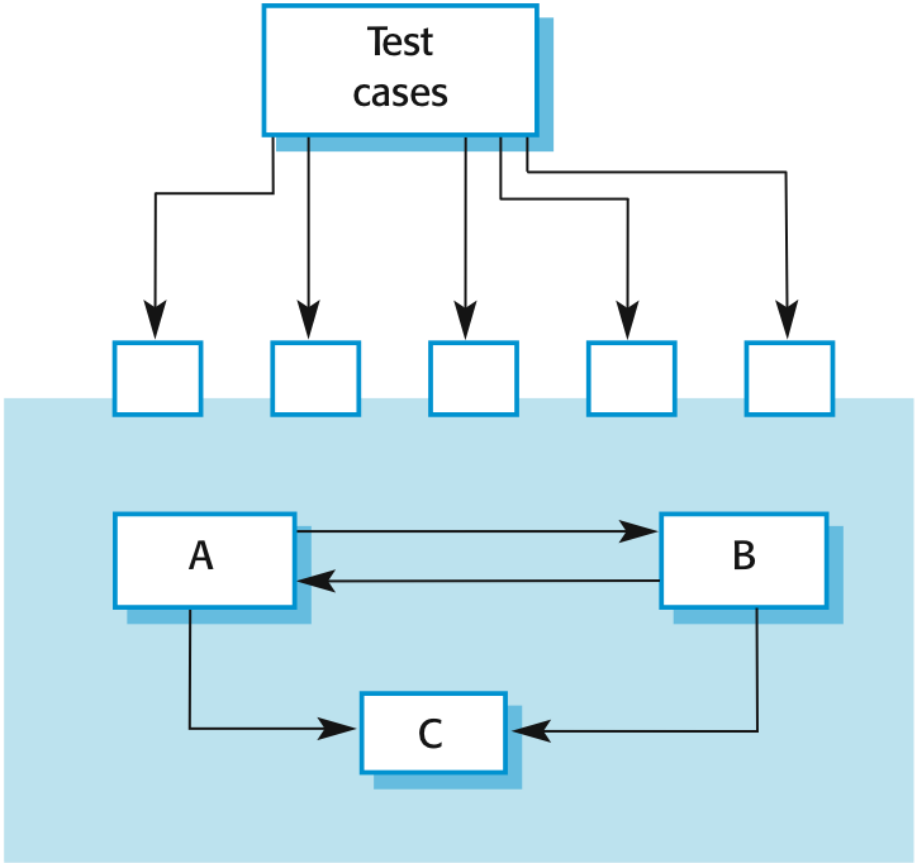
- ✧ Choose inputs that force the system to generate all error messages
- ✧ Design inputs that cause input buffers to overflow
- ✧ Repeat the same input or series of inputs numerous times
- ✧ Force invalid outputs to be generated
- ✧ Force computation results to be too large or too small

Component testing

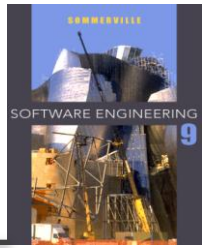


- Software components are often composite components that are made up of several interacting objects
- Functionality of a component is accessed through a defined interface
- Focus on showing that the component interface behaves according to its specification
  - Assume that unit tests on the individual objects within the component have been completed

Interface testing

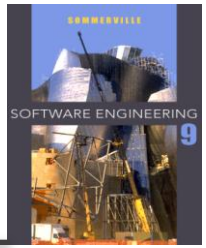


Interface testing



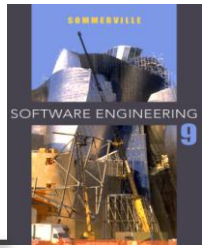
- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces (conflicts with specification)
- Interface types
  - Parameter interfaces Data passed from one method or procedure to another (function call)
  - Shared memory interfaces A block of memory is shared between procedures or functions (e.g. parallel threads memory pools)
  - Procedural interfaces A sub-system encapsulates a set of procedures to be called by other sub-systems (objects, API)
  - Message passing interfaces Sub-systems request services from other sub-systems (client-server, web browser)

Interface errors



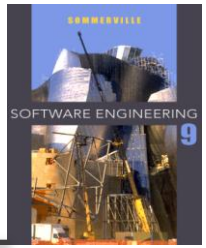
- Interface misuse
  - A calling component calls another component and makes an error in its use of its interface, e.g. parameters in the wrong order
- Interface misunderstanding
  - A calling component embeds assumptions about the behaviour of the called component which are incorrect (e.g., assume an input array is ordered, and in fact it is not)
- Timing errors
  - The called and the calling component operate at different speeds and out-of-date information is accessed

Interface testing guidelines



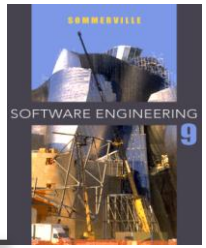
- Design tests so that parameters to a called procedure are at the extreme ends of their ranges
- Always test pointer parameters with null pointers
- Design tests which cause the component to fail
- Use stress testing in message passing systems
- In shared memory systems, vary the order in which components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

System testing



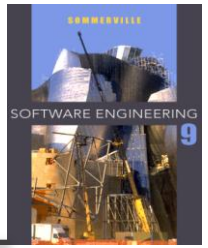
- System testing during development involves integrating components to create a version of the system and then testing the integrated system
- The focus is testing interactions between components
- Checks that components are compatible, interact correctly and transfer data correctly across their interfaces
- Tests the emergent behavior (behavior not explicitly described by the components, and is unexpected)

System and component testing



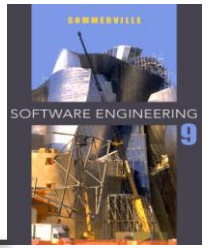
- Reusable components that have been separately developed AND off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
  - In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

Testing scenario



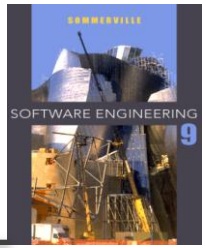
A student in Scotland is studying American History and has been asked to write a paper on ÕFrontier mentality in the American West from 1840 to 1880Õ.To do this, she needs to find sources from a range of libraries. She logs on to the LIBSYS system and uses the search facility to discover if she can access original documents from that time. She discovers sources in various US university libraries and downloads copies of some of these. However, for one document, she needs to have confirmation from her university that she is a genuine student and that use is for non-commercial purposes. The student then uses the facility in LIBSYS that can request such permission and registers her request. If granted, the document will be downloaded to the registered libraryÕs server and printed for her. She receives a message from LIBSYS telling her that she will receive an e-mail message when the printed document is available for collection.

System tests



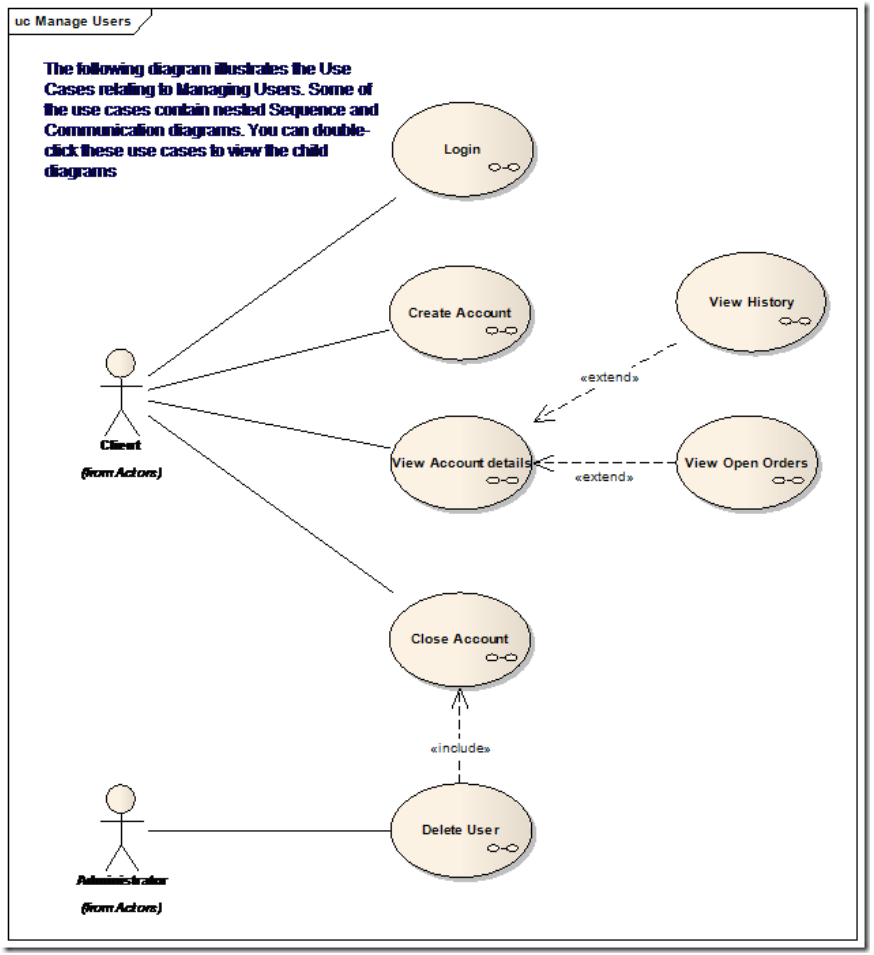
1. Test the login mechanism using correct and incorrect logins to check that valid users are accepted and invalid users are rejected.
2. Test the search facility using different queries against known sources to check that the search mechanism is actually finding documents.
3. Test the system presentation facility to check that information about documents is displayed properly.
4. Test the mechanism to request permission for downloading.
5. Test the e-mail response indicating that the downloaded document is available.

Use-case testing



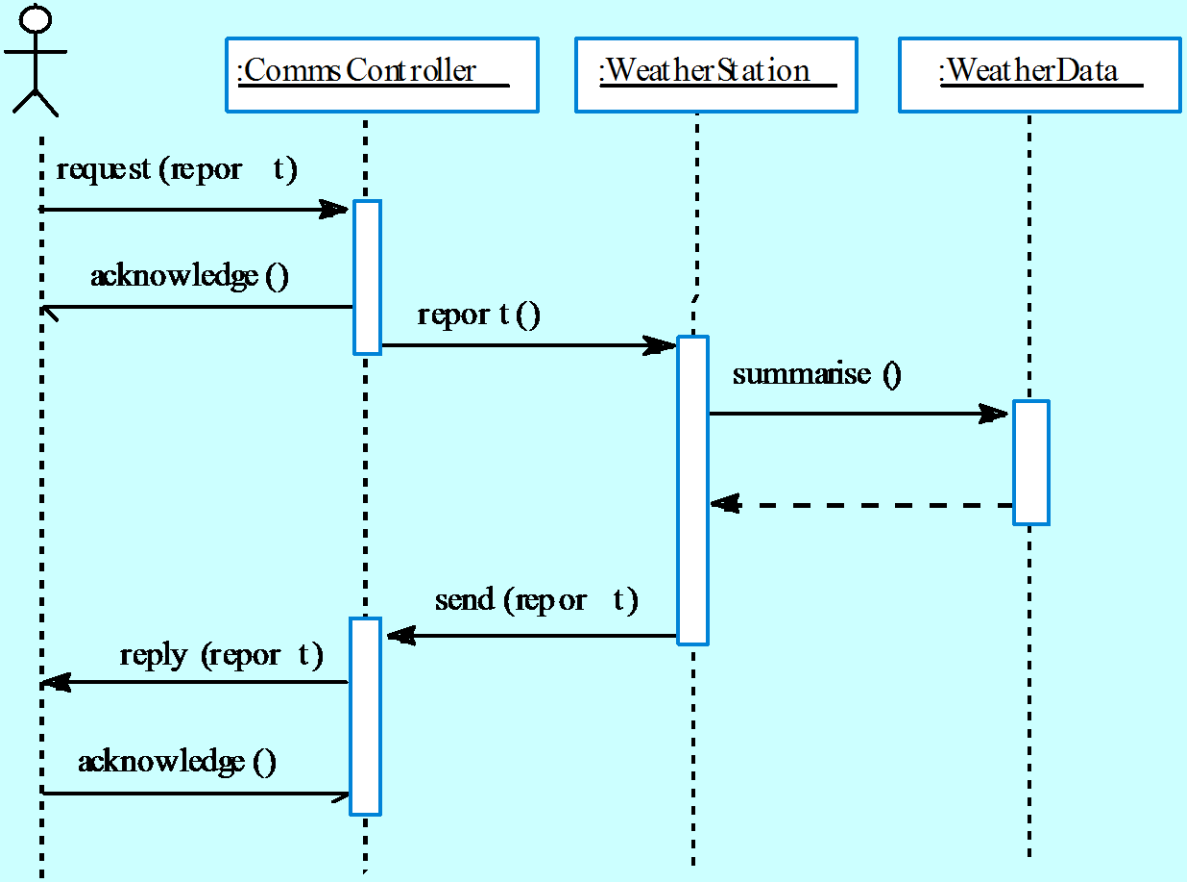
- The use cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur
- The sequence diagrams associated with the use case documents the components and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

Use-case testing

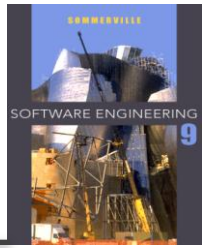


http://andrewtokeley.net/archive/2008/03/24/enterprise-architect-for-testers.aspx

Collect weather data sequence chart

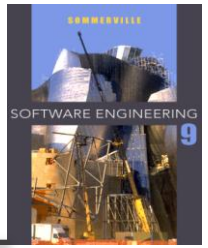


Performance testing



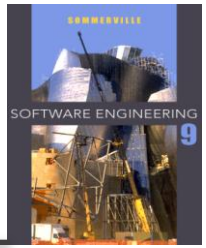
- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

Stress testing



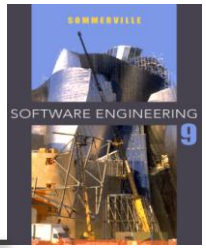
- Exercises the system beyond its maximum design load. Stressing the system often causes defects to come to light.
- Stressing the system test failure behaviour.. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data.
- Stress testing is particularly relevant to distributed systems that can exhibit severe degradation as a network becomes overloaded.

Testing policies



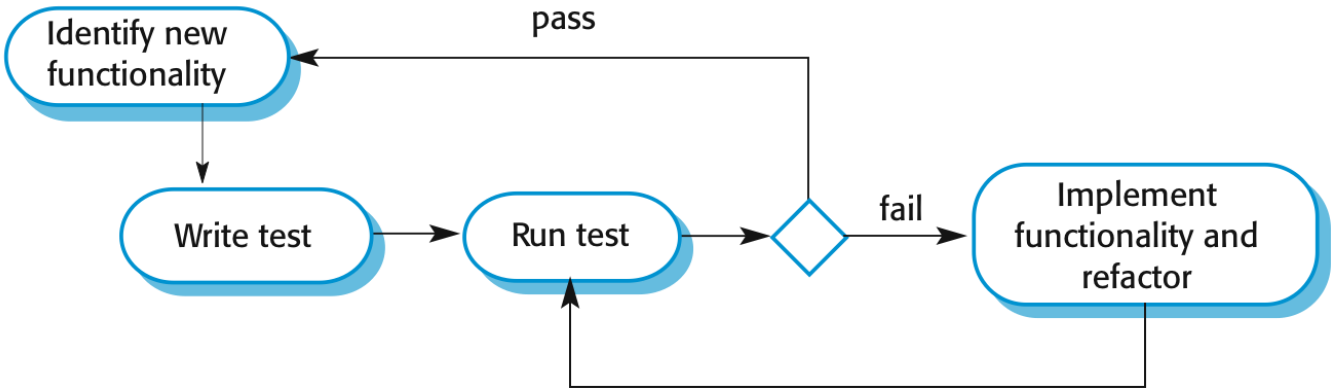
- Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- Examples of testing policies:
  - All system functions that are accessed through menus should be tested
  - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - Where user input is provided, all functions must be tested with both correct and incorrect input

Test-driven development

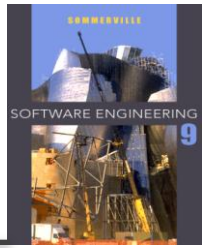


- Test-driven development (TDD) is an approach to program development in which you create tests before writing code
- You develop code incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

Test-driven development

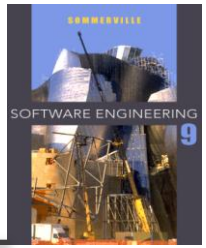


TDD process activities



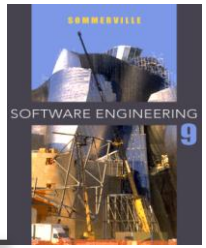
1. Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
2. Write a test for this functionality and implement this as an automated test
3. Run the test, along with all other tests that have been implemented. Initially, 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



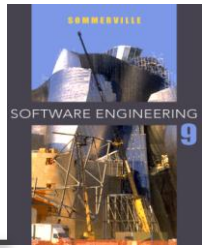
- ✧ Code coverage
  - Every code segment that you write has at least one associated test so all code written has at least one test.
- ✧ Regression testing
  - A regression test suite is developed incrementally as a program is developed
- ✧ Simplified debugging
  - When a test fails, it should be obvious 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



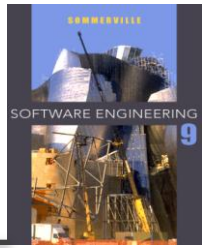
- Regression testing is testing the system to check that changes have not 'broken' previously working code.
- Manual regression testing is expensive
- Automated regression testing is relatively simple and straightforward
- All tests are rerun every time a change is made to the program
- Tests must run 'successfully' before the change is committed

Release testing



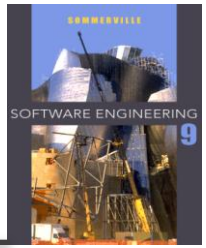
- Release testing is the process of testing a particular release of a system
- The primary goal is to convince the supplier of the system that it is good enough for release.
  - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability,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

Release testing and system testing



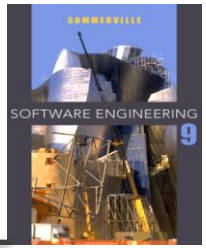
- ✧ Forms of release testing:
  - Requirements-based testing
  - Scenario-based testing
- ✧ Release testing is a form of system testing
- ✧ Important differences:
  - A separate team that has not been involved in the system development, should be responsible for release testing
  - System testing by 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*).

Requirements based testing



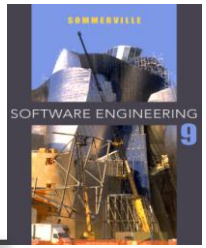
- Requirements-based testing involves examining each requirement and developing a test or tests for it

Performance testing



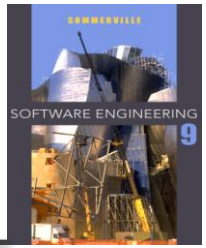
- Test the emergent properties of a system, such as performance and reliability
- Tests should reflect the profile of use of the system
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable
- Stress testing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

User testing



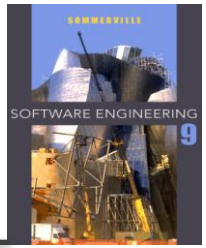
- User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing
- User testing is essential, even when comprehensive system and release testing have been carried out
  - The reason for this is that influences from the user's working 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



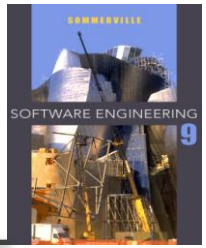
- ✧ Alpha testing
  - Users of the software work with the development team to test the software at the developer's site
- ✧ Beta testing
  - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers
- ✧ Acceptance 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.

Key points



- Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.
- Development testing includes unit testing, component testing, and system testing

Key points



- When testing software, you should try to 'break' the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.
- Wherever possible, you should write automated tests
- Test-driven development is an approach to development where tests are written before the code to be tested.
- Scenario testing involves inventing a typical usage scenario and using this to derive test cases.
- Acceptance testing is a **user** testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment (usually for custom software)