



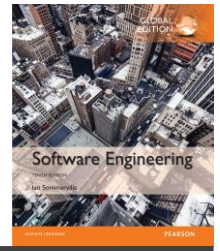
Chapter 8 – Software Testing

Topics covered



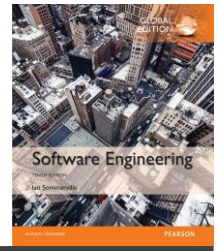
- ✧ Development testing
- ✧ Test-driven development
- ✧ Release testing
- ✧ User testing

Program testing



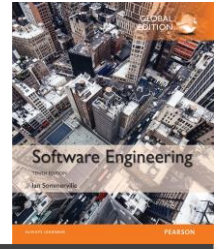
- ✧ Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.
- ✧ When you test software, you execute a program using **artificial data**.
- ✧ You check the results of the test run for **errors**, **anomalies** or information 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** (V&V) process, which also includes static validation techniques.

Program testing goals



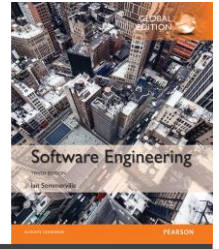
- ✧ To **demonstrate** to the developer and the customer that the software meets its requirements.
 - For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- ✧ To **discover** situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
 - 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. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

Testing process goals



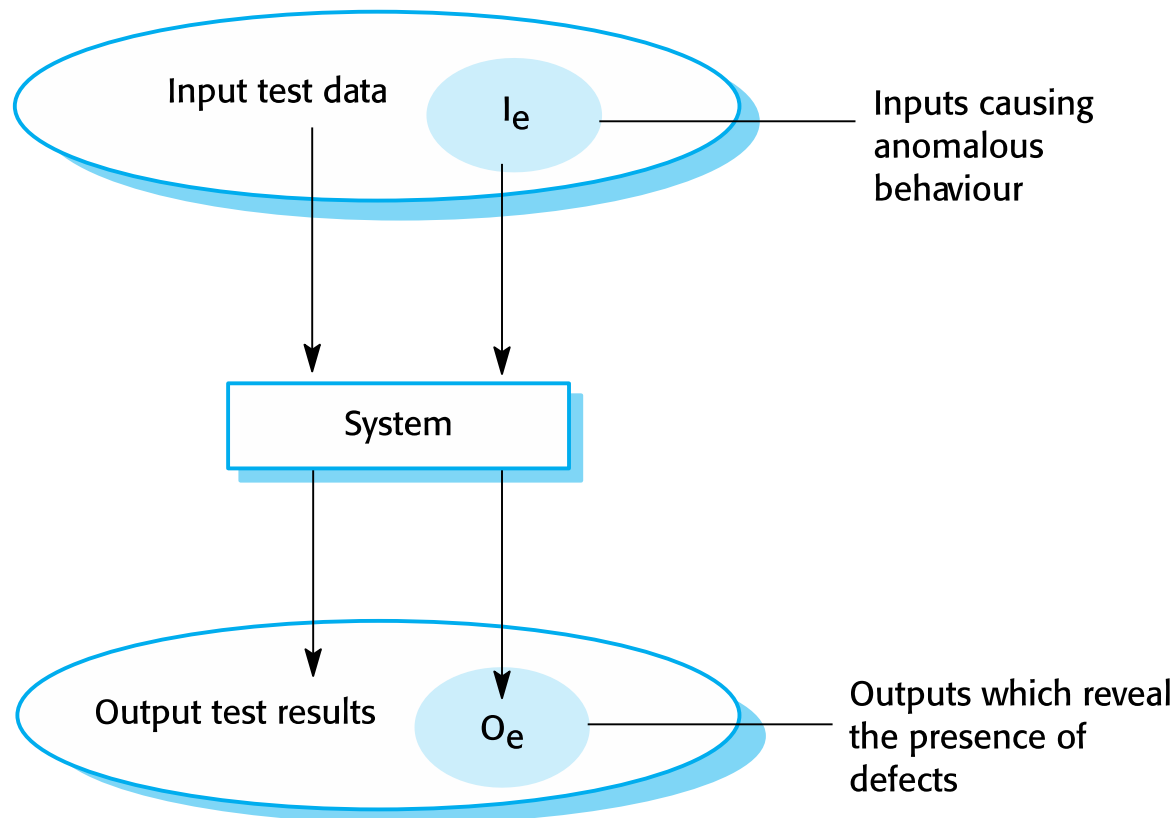
✧ Validation testing

- 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 behaviour 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.

An input-output model of program testing



Verification vs validation (V&V)



✧ **Verification:**

"Are we building the product right".

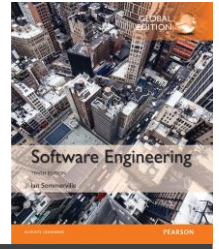
✧ The software should **conform to its specification.**

✧ **Validation:**

"Are we building the right product".

✧ The software should **do what the user really requires.**

V & V confidence



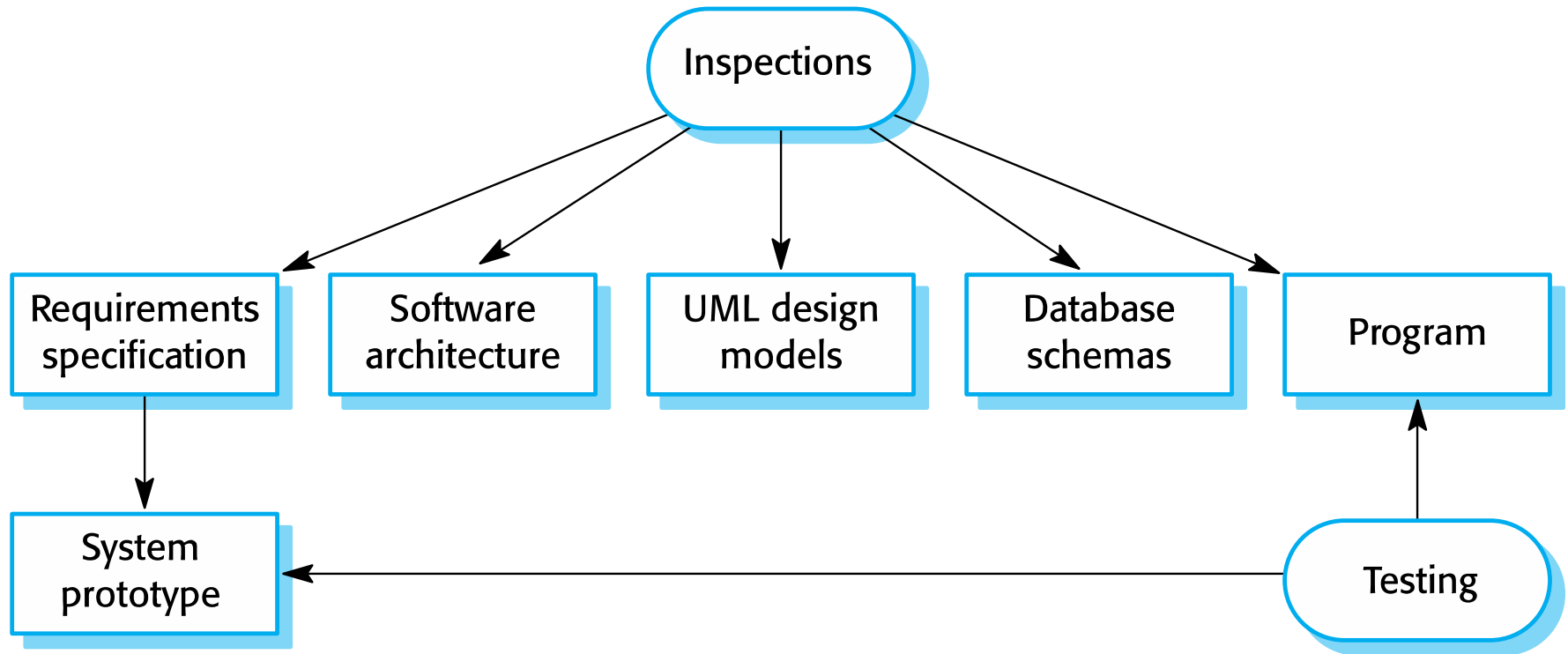
- ✧ Aim of V & V is to establish confidence that the system is 'fit for purpose'.
 - The system must good enough for its intended use
- ✧ Depends on system's purpose, user expectations and marketing environment
 - **Software purpose**
 - The level of confidence depends on how critical the software is to an organisation.
 - **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

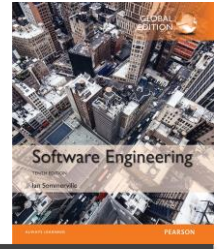


- ✧ **Software inspections** Concerned with analysis of the static system representation to discover problems (**static verification**)
 - May be supplement by tool-based document and **code analysis**.
 - Discussed in Chapter 24.
- ✧ **Software testing** Concerned with exercising and observing product behaviour (**dynamic verification**)
 - The system is executed with **test data** and its **operational behaviour** is observed.

Inspections and testing



Software inspections



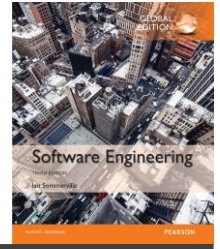
- ✧ These involve people examining the **source representation** with the aim of discovering anomalies and defects.
- ✧ Inspections not require execution of a system so **may be used before implementation**.
- ✧ They may be **applied to any representation of the system** (requirements, design, configuration data, test data, etc.).
- ✧ They have been shown to be **an effective technique for discovering program errors**.
 - 60% of the errors in a program can be detected using informal program inspections (Fagan 1986)

Advantages of inspections



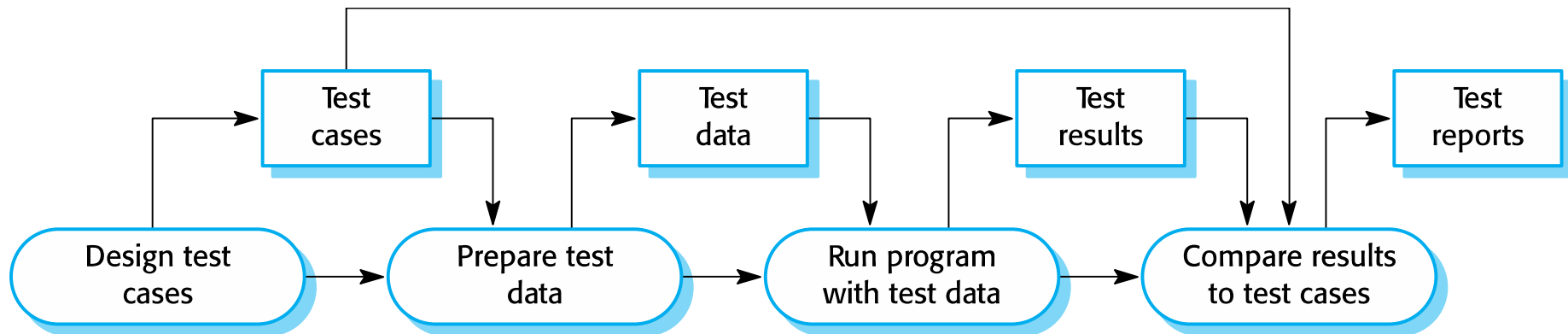
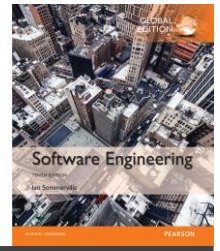
- ✧ During testing, errors can mask (hide) other errors.
Because inspection is a static process, you don't have to be concerned with interactions between errors.
 - A single inspection session can discover many errors in a system
- ✧ Incomplete versions of a system can be inspected without additional 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, unexpected interactions between different parts of a program, timing problems, etc.

A model of the software testing process



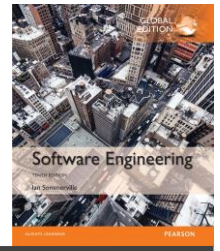
Stages of testing



- ✧ **Development testing**, where the system is tested during development to discover bugs and defects.
 - Is primary a **defect testing** process
 - System designers and programmers are likely to be involved in the testing process
- ✧ **Release testing**, where a **separate testing team** test a **complete version** of the system before it is released to users.
 - The aim of release testing is to check that the system meets the requirements of system stakeholders
- ✧ **User testing**, where users or potential users of a system **test the system in their own environment**.
 - E.g., acceptance testing

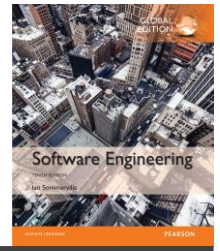
Development testing

Development testing



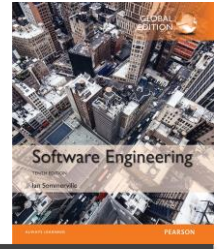
- ✧ Development testing includes all testing activities that are carried out by the team developing the system.
 - **Unit testing**, where individual **program units or object classes** are tested. Unit testing should focus on testing the functionality of objects or methods.
 - **Component testing**, where **several individual units are integrated to create composite components**. Component testing should focus on testing component interfaces.
 - **System testing**, where 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



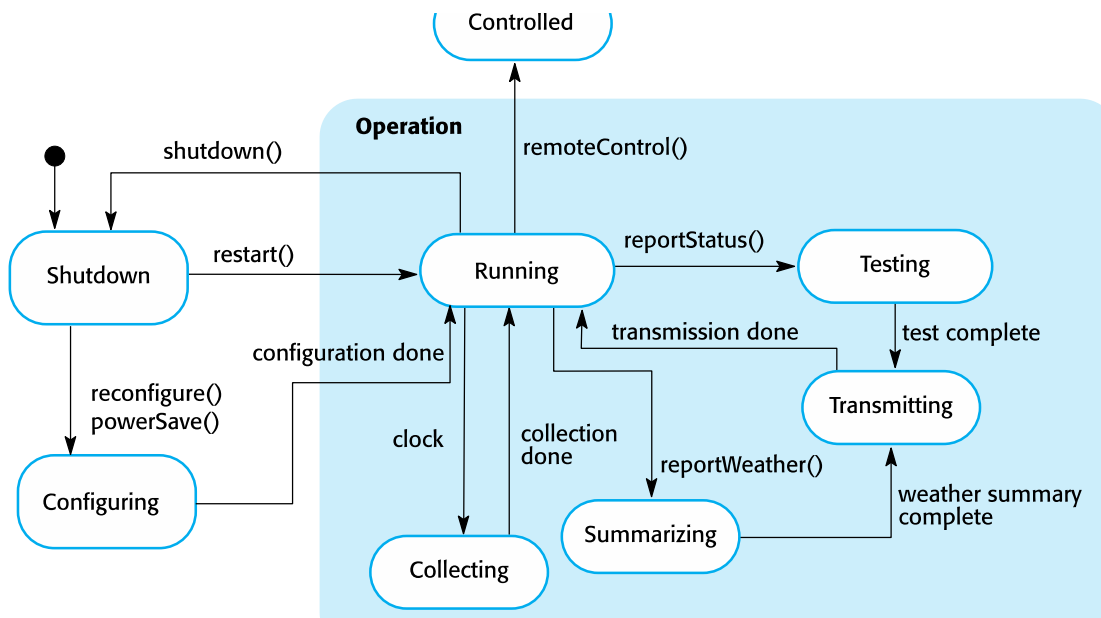
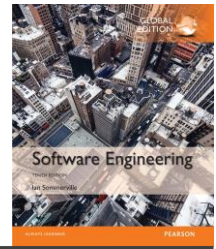
- ✧ Unit testing is the process of testing individual components in isolation.
- ✧ It is a **defect testing** process.
- ✧ Units may be:
 - Individual **functions or methods** within an object
 - **Object classes** with several attributes and methods
 - **Composite components** with defined interfaces 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.
 - The operation that is inherited may make assumptions about other operations and attributes. These may NOT be valid in some subclasses that inherit the operation
 - Thus, the inherited operation has to be tested in all of the contexts where it is used

The weather station object interface

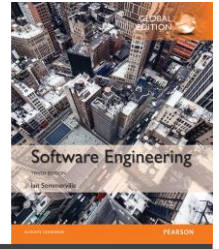


WeatherStation

identifier

reportWeather ()
reportStatus ()
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

Weather station testing



- ✧ Need to define test cases for reportWeather, reportStatus, ..., reconfigure, restart and shutdown.
- ✧ 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

Automated testing



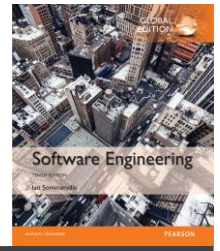
- ✧ Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- ✧ In automated unit testing, you make use of a **test automation framework** (such as JUnit) to write and run your program tests.
- ✧ Unit testing frameworks provide generic test classes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

Automated test components



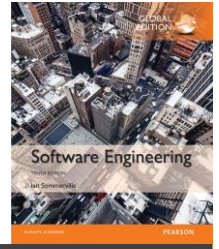
- ✧ A **setup part**, where you initialize the system with the test case, namely the inputs and expected outputs.
- ✧ A **call part**, where you call the object or method to be tested.
- ✧ An **assertion part** where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.
- ✧ Sometimes the object that you are testing has dependencies on other objects that may **not have been written** or which **slow down the testing process** if they are used
 - **Mock object:** objects with the same interface as the external objects being used that simulate its functionality (or simulate abnormal operations or rare events). e.g., database mock objects

Choosing unit test cases



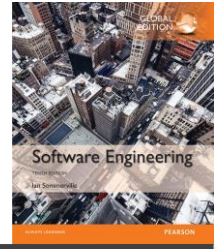
- ✧ The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
- ✧ If there are defects in the component, these should be revealed by test cases.
- ✧ This leads to 2 types of unit test case:
 - The first of these should **reflect normal 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.

Testing strategies



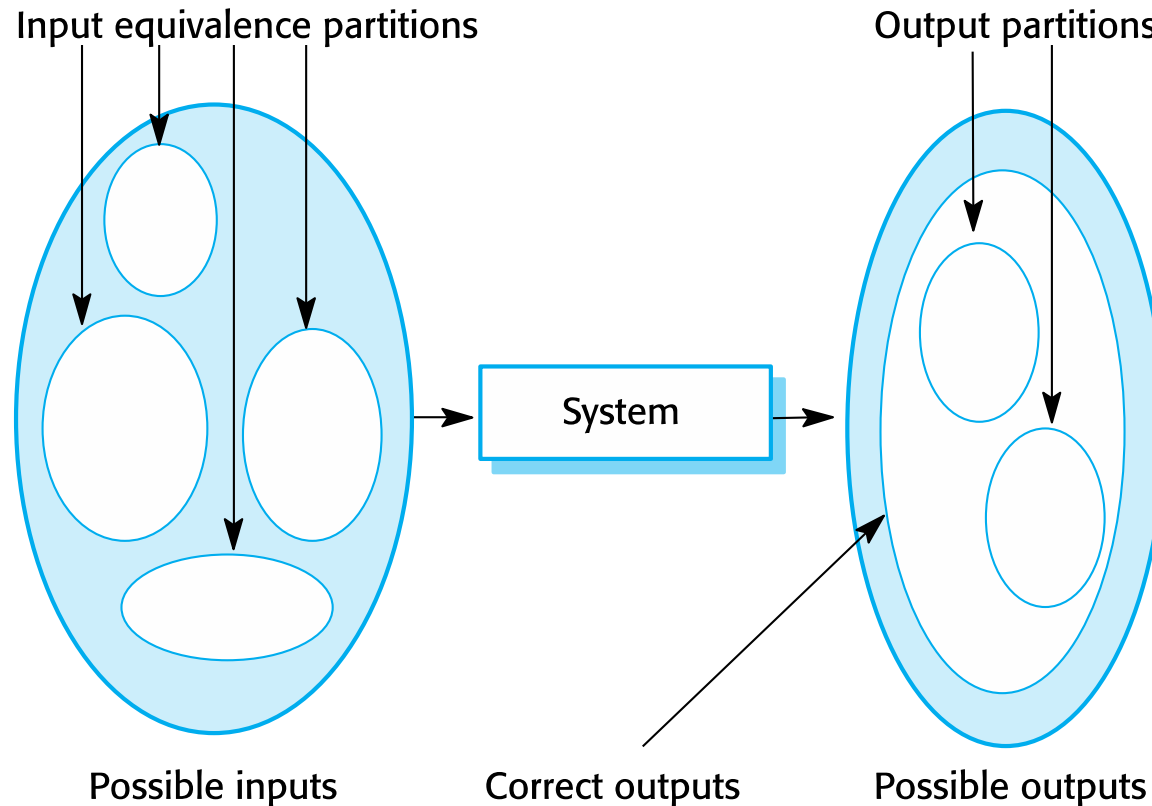
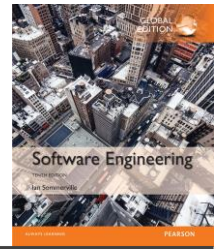
- ✧ **Partition testing**, where you identify groups of inputs that have common characteristics and should be processed in the same way. (**equivalence partitioning**)
 - You should choose tests from within each of these groups.
- ✧ **Guideline-based testing**, where you use testing guidelines to choose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

Partition testing



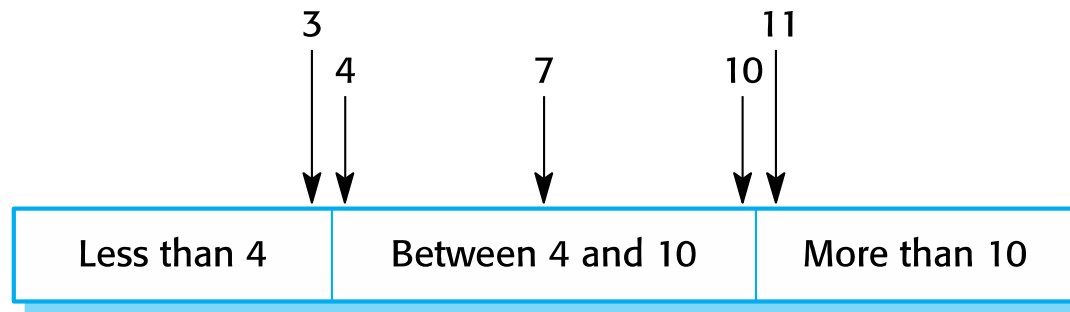
- ✧ **Input data** and **output results** often fall into different classes with common characteristics where all members of a class are related and be processed by the program under test in the same way
- ✧ Each of these **classes** is an **equivalence partition** or **domain** where the program behaves in an equivalent way for each class member.
- ✧ **Test cases should be chosen from each partition.**
- ✧ Partition testing can be used to design test cases for both systems and components.
- ✧ **A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition. (boundary value analysis, BAV)**

Equivalence partitioning

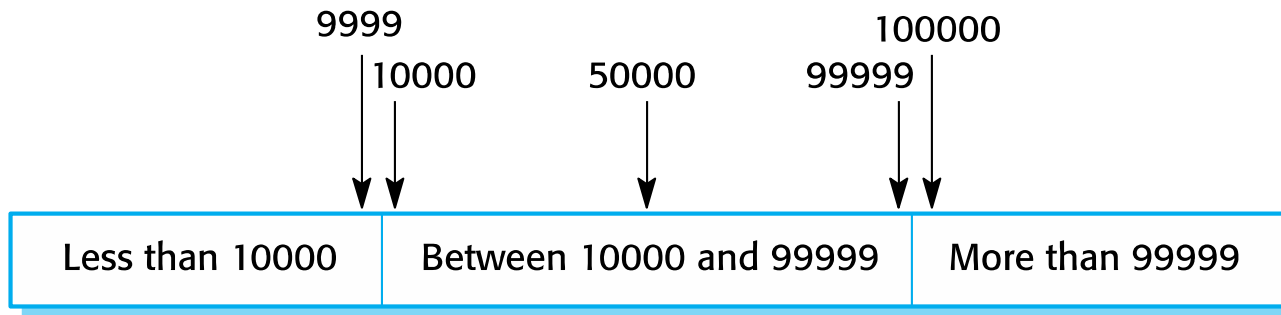


Output equivalence partitions: are partitions within which all the outputs have something in common

Equivalence partitions

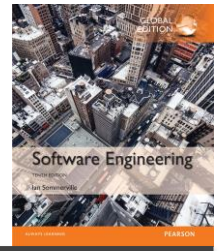


Number of input values



Input values

Black-box vs. white-box testing



✧ Black-box testing technique

- Use the specification of a system to design test cases
- E.g., use the specification of a system to identify equivalence partitions

✧ White-box testing technique

- Use the information of code to design test cases
- E.g., use the knowledge of the code that may include exceptions to handle incorrect inputs to identify “exception partitions” – different ranges where the same exception handling should be applied

Testing guidelines (sequences)



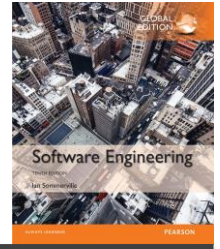
- ✧ Guidelines encapsulate knowledge of what kinds of test cases are effective for discovering errors.
- ✧ Example of testing guidelines to test programs with sequences, arrays, or lists
 - Test software with sequences which have only a single value.
 - Use sequences of different sizes in different tests.
 - Derive tests so that the first, middle and last elements of the sequence are accessed.
 - Test with sequences of zero length.

General testing guidelines (Whittaker 2002)



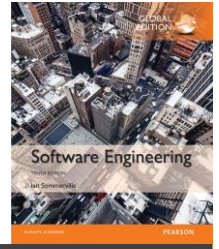
- ✧ Choose inputs that force the system to generate all error messages
- ✧ Design inputs that cause input buffers to overflow
- ✧ Repeat the same input or series of inputs numerous times
- ✧ Force invalid outputs to be generated
- ✧ Force computation results to be too large or too small.

Component testing



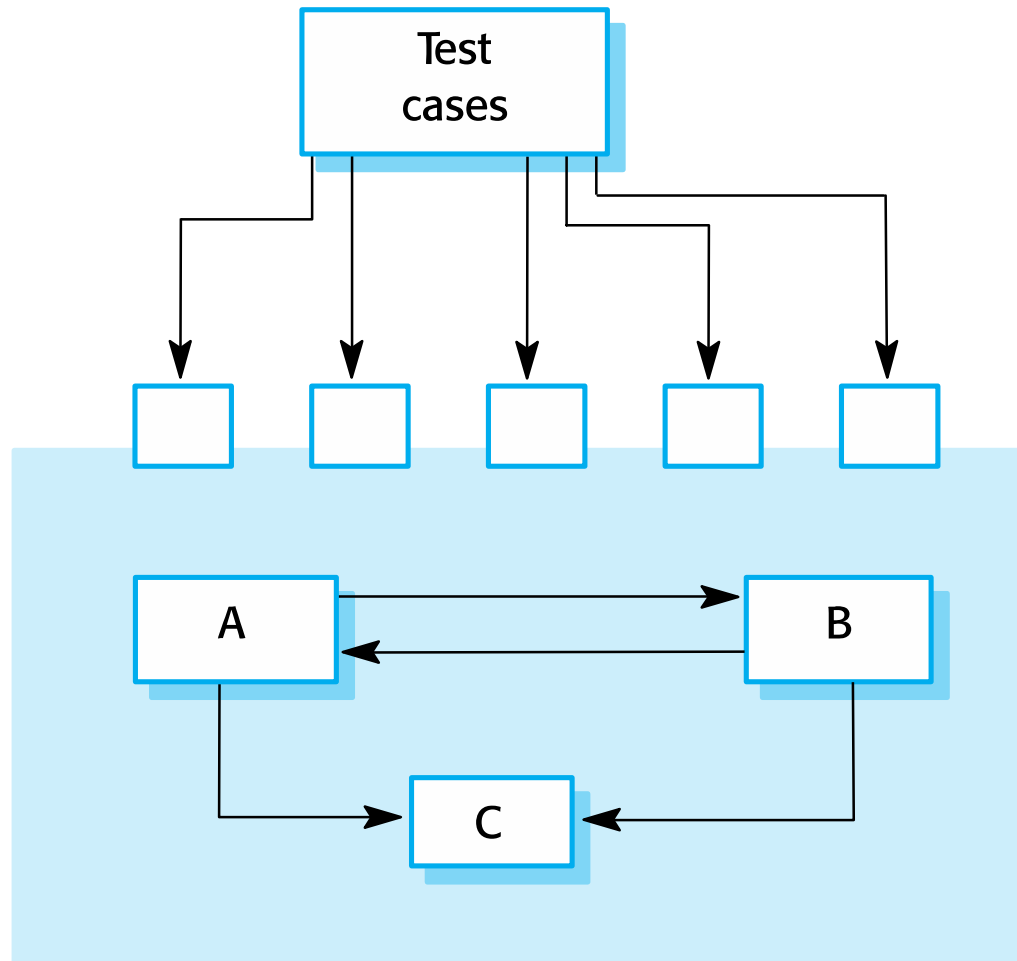
- ✧ Software components are often composite components that are made up of several interacting objects.
 - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- ✧ You access the functionality of these objects through 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**.
 - These errors result from **interactions between the objects** in the component
- ✧ Interface types
 - **Parameter interfaces** Data (or function references) 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



Interface errors



✧ Interface misuse

- A calling component calls another component and makes an error in its use of its interface e.g. parameters of the wrong **type**, parameters in the wrong **order**, or the wrong **number** of parameters.

✧ Interface misunderstanding

- A calling component embeds assumptions about the behaviour 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.

Interface testing



- ✧ Testing for interface is difficult because some interface faults may only manifest themselves under unusual conditions
 - For example, an object implements a queue as a fixed-length data structure. A calling object may assume that the queue is implemented as an infinite data structure and may not check for queue overflow when an item is entered
- ✧ A further problem may arise because of interactions between faults in different modules or objects
 - Faults in one object may only be detected when some other object behaves in an unexpected way
- ✧ **Inspections and reviews** can sometimes be more cost effective than testing for discovering interface errors

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.
 - Differing failure assumptions are one of the most common specification misunderstanding
- ✧ Use stress testing in message passing systems.
 - An effective way of revealing timing problems
- ✧ In shared memory systems, vary the order in which components are activated.
 - May reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed

System testing



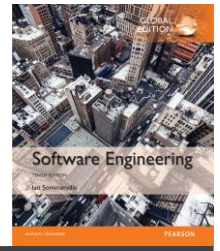
- ✧ System testing during development involves integrating components to create **a version of the system** and then testing the integrated system.
- ✧ The focus in system testing is testing the interactions between components.
- ✧ System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- ✧ System testing tests the **emergent behaviour of a system**.
 - Some elements of system functionality only become obvious when you put the components together

Differences of 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.

Use-case testing

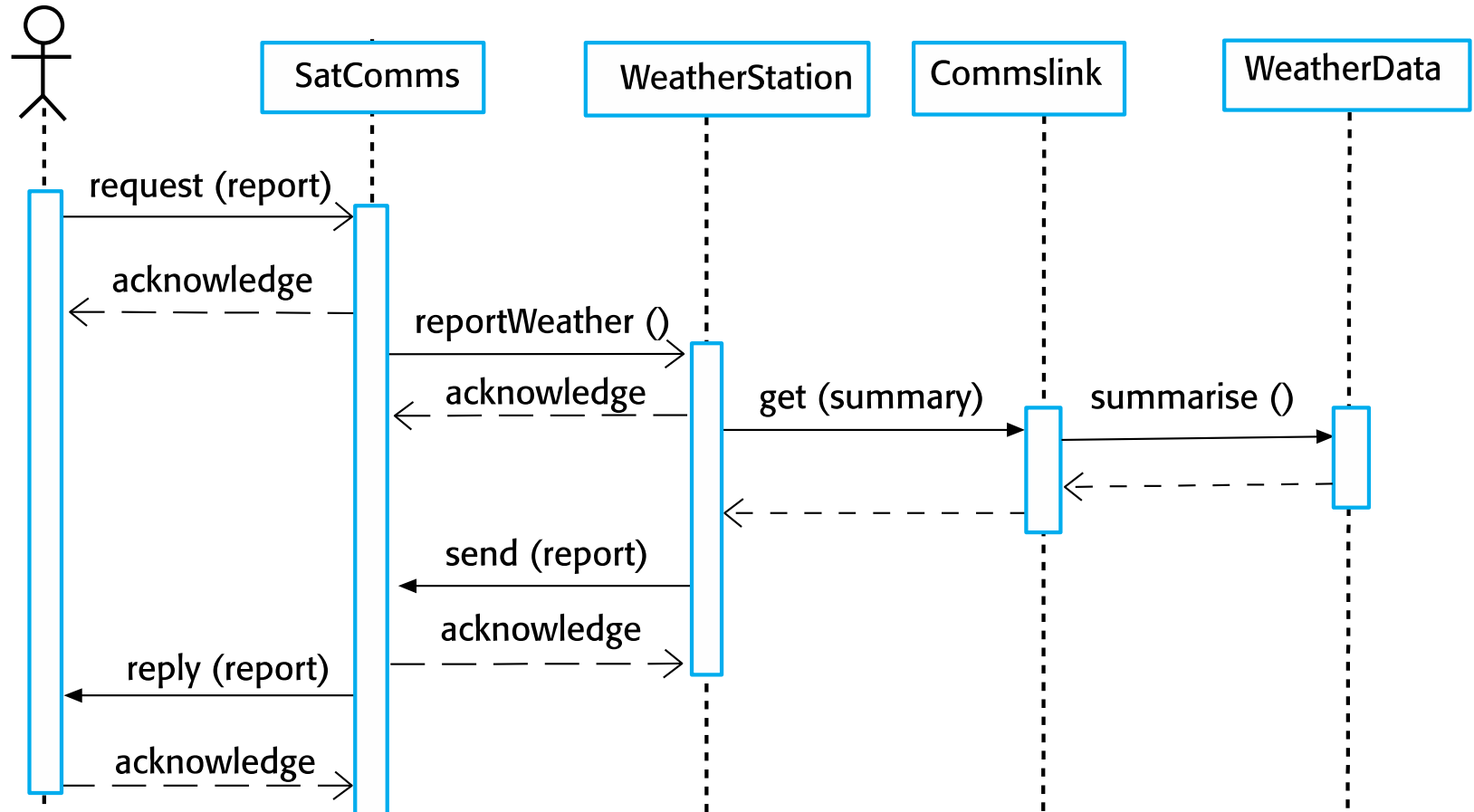


- ✧ The **use-cases** developed to identify **system interactions** can be used as a basis for **system testing**.
- ✧ Each use case usually involves several system components **so testing the use case forces these interactions to occur**.
- ✧ The sequence diagrams associated with the use case documents the components and interactions that are being tested.
 - You can use sequence diagram to identify operations that will be tested and to help design the test cases to execute the tests
 - The sequence shows what inputs are required and what outputs are created

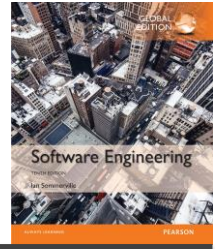
Collect weather data sequence chart



information system



Test cases derived from sequence diagram



- ✧ An input of a request for a report should have an associated acknowledgement. A report should ultimately be returned from the request.
 - You should create summarized data that can be used to check that the report is correctly organized.
- ✧ An input request for a report to WeatherStation results in a summarized report being generated.
 - Can be tested by creating raw data corresponding to the summary that you have prepared for the test of SatComms and checking that the WeatherStation object correctly produces this summary. This raw data is also used to test the WeatherData object.

Testing policies

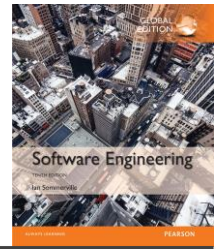


- ✧ Exhaustive system testing is impossible so testing policies which define the required system **test coverage** may be developed.
 - Ideally, software companies should have **policies** for choosing a subset of possible test cases
- ✧ Examples of testing policies:
 - **All system functions** that 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.**



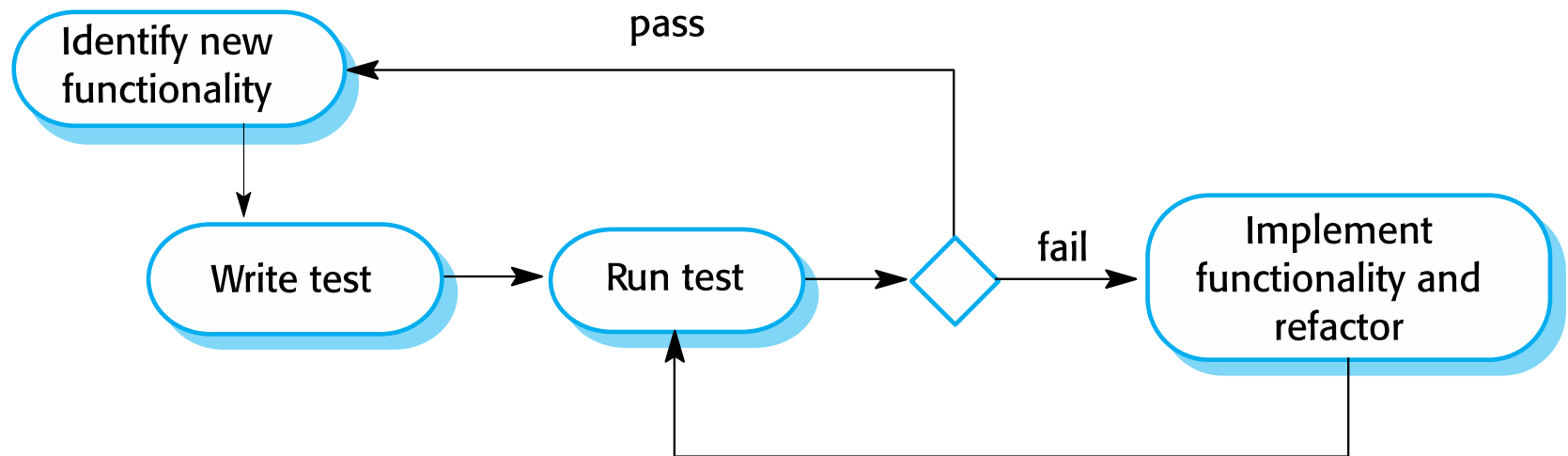
Test-driven development

Test-driven development



- ✧ 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.
- ✧ TDD helps programmers clarify their ideas of what a code segment is **actually supposed to do**. To write a test, you need to understand what is intended, as this **understanding** makes it easier to write the **required code**.

Test-driven development

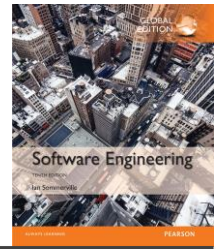


TDD process activities



- ✧ 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. **You can always run regression tests to check that changes to program have not introduced new bugs.**

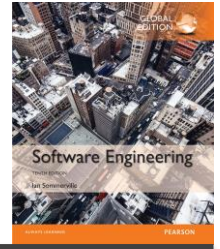
✧ Simplified debugging

- 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. Reading the tests can make it easier to understand the code

Regression testing



- ✧ One of the most important benefits of TDD is that it reduces the costs of 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**.



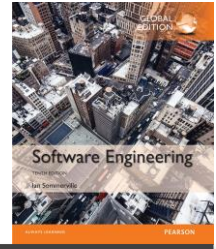
Release testing

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.

- Is the process of testing a particular release of a system that is intended for use outside of the development team

✧ Important differences (between release and system testing):

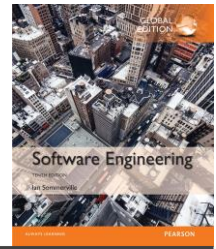
- A separate team that has not been involved in the system development, should be responsible for release testing.
- System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements 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.
 - A general principle of good requirements engineering practice is that requirements should be testable
- ✧ **Requirement-based testing is validation** rather than defect testing
 - Try to demonstrate that the system has properly implemented its requirements
- ✧ Mentcare system requirements:
 - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
 - If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

Requirements tests



- ✧ 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.**
- ✧ 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.
- ✧ Set up a patient record in which **allergies to two or more drugs** are recorded. Prescribe both of these drugs **separately** and **check that the correct warning for each drug is issued.**
- ✧ Prescribe **two drugs** that the patient is allergic to. **Check that two warnings are correctly issued.**
- ✧ Prescribe a drug that **issues a warning and overrule that warning**. **Check that the system requires the user to provide information** explaining why the warning was overruled.

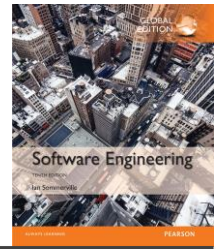
Testing **a requirement** usually requires to write **several tests** to ensure that you have coverage of the requirement. You should also maintain **traceability** records of your requirement-based testing.

Scenario testing



- ✧ 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.
 - A **scenario is a story** that describes one way in which the system might be used
 - Scenarios should be **realistic** and real system users should be able to relate to them.
- ✧ A **scenario test should be a narrative story that is credible and fairly complex**. It should motivate stakeholders, that is, they should relate to the scenario and believe that it is important that the system passes the test.

A usage scenario for the Mentcare system



George is a nurse who specializes in mental healthcare. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients that he visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. George looks up Jim's record and is prompted for his key phrase to decrypt the record. He checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so he notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. Jim agrees so George enters a prompt to call him when he gets back to the clinic to make an appointment with a physician. George ends the consultation and the system re-encrypts Jim's record.

After, finishing his consultations, George returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for George of those patients who He has to contact for follow-up information and make clinic appointments.

Features tested by scenario



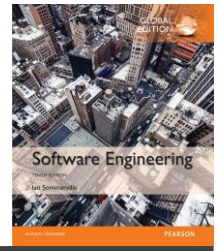
- ✧ Authentication by logging on to the system.
- ✧ Downloading and uploading of specified patient records to a laptop.
- ✧ Home visit scheduling.
- ✧ Encryption and decryption of patient records on a mobile device.
- ✧ Record retrieval and modification.
- ✧ Links with the drugs database that maintains side-effect information.
- ✧ The system for call prompting.

Scenario testing



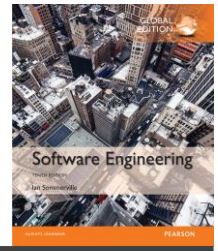
- ✧ If you are a release tester, you run through this scenario, playing the role of George and observing how the system behaves in response to different inputs.
 - You should carefully note problems that arise, including performance problem.
- ✧ When you use a scenario-based approach, you are normally testing several requirements within the same scenario. Therefore, as well as checking individual requirements, you are also checking combinations of requirements do not cause problems

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**.
 - **Operational profile**: is a set of tests that reflect the actual mix of work that will be handled by the system
- ✧ **Performance tests** usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
 - Is concerned **both** with demonstrating that **the system meets its requirements** and **discovering problems and defects in the system**

Performance testing



- ✧ **Stress testing** is a form of performance testing where the system is deliberately **overloaded to test its failure behaviour**.
 - It tests the failure behavior of the system.
 - **Stress testing checks that overloading the system causes it to “fail-soft” rather than collapse under its load**
 - It stresses the system and may cause **defects** to come to light that **would not normally be discovered**.
- ✧ Stress testing is particularly relevant **distributed systems** based on a network of processors
- ✧ **Stress testing helps you discover when the degradation begins so that you can add checks to the system to reject transactions beyond this point**



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**.

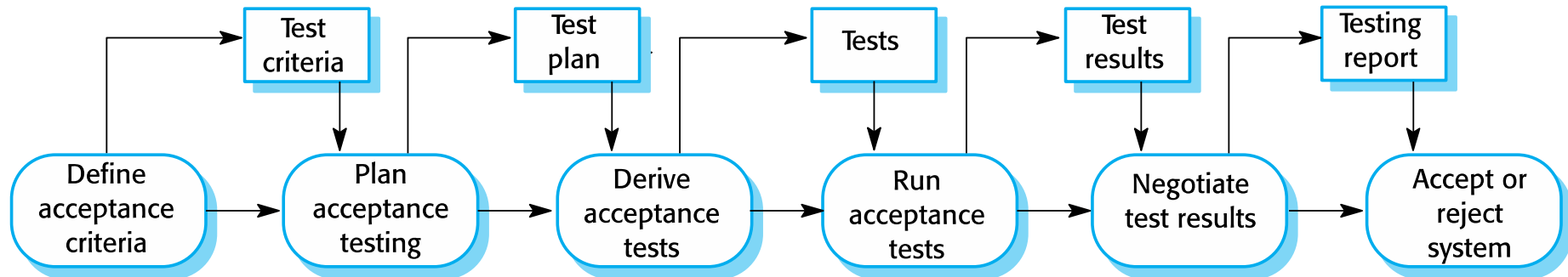
✧ Beta testing

- 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.
- Is mostly used for software products that are used in many different environments

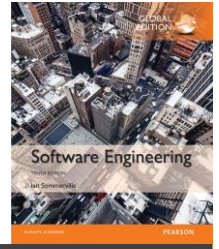
✧ Acceptance testing

- 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

- The acceptance criteria should be part of the system contract and be agreed between the customer and the developer

✧ Plan acceptance testing

- Decide on the resource, time, and budget for acceptance testing and establish a testing schedule; discuss the required coverage of the requirements and test order of the features

✧ Derive acceptance tests

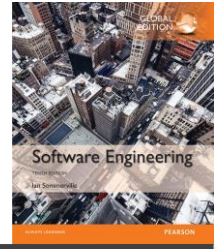
- Aim to test both the functional and non-functional characteristics of the system and should, ideally, provide complete coverage of the system requirements

✧ Run acceptance tests

✧ Negotiate test results

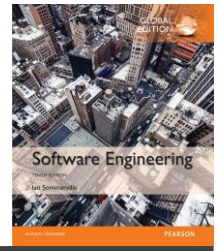
✧ Reject/accept system

Agile methods and acceptance testing



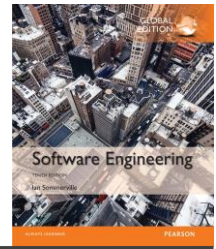
- ✧ In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- ✧ Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- ✧ There is no separate acceptance testing process.
- ✧ Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.
 - If not, acceptance testing may require groups of end-users to use and test the system

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, in which you test individual objects and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems.

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**. The tests are embedded in a program that can be run every time a change is made to a system.
- ✧ **Test-first 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.