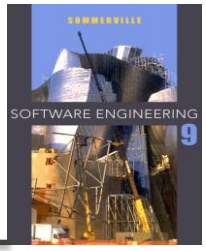
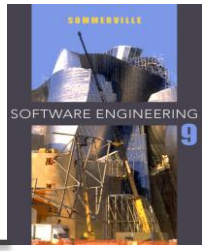



# Chapter 8 – Software Testing

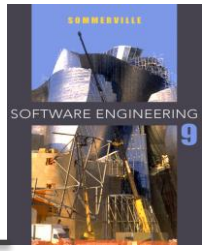
Lecture 1

# Topics covered



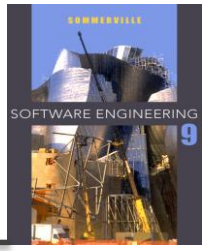

- ✧ Development testing
- ✧ Test-driven development
- ✧ Release testing
- ✧ User testing

# Program testing



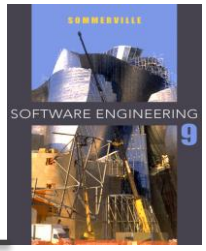

- Testing is intended to show that a program does what it is intended to do and to **discover program defects** before it is put into use.
- When you test software, you execute a program using **artificial data**.
- You check the results of the **test run for errors, anomalies** or information about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

# Program testing goals



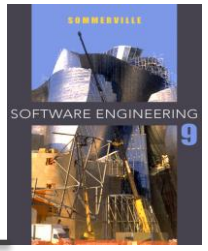

- **To demonstrate to the developer and the customer** that the software meets its requirements.
  - For **custom software**, this means that there should be **at least one test for every requirement** in the requirements document. **For generic software products**, it means that there should be **tests 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 crashe**s, **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**.
- Of course, there is **no definite boundary between these two approaches** to testing.
  - During validation testing, you will **find defects** in the system;
  - **During defect testing**, **some of the tests will show that the pro-gram meets its requirements**.

# Testing process goals



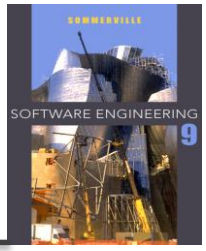

✧ Validation testing

- 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

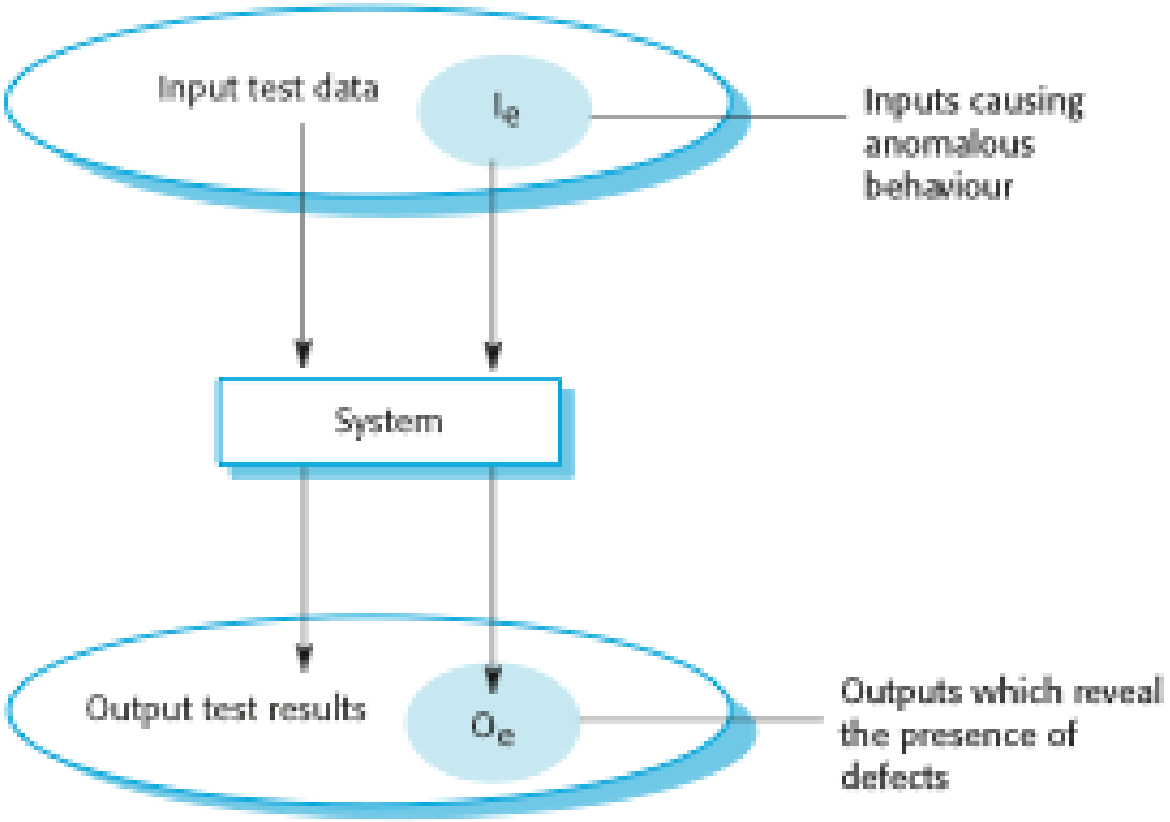
- To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification
- **A successful test** is a test that makes the **system perform incorrectly and so exposes a defect in the system**.

# The difference between validation testing and defect Testing

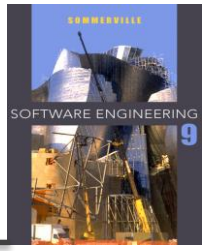



- The following Figure may help to explain the differences between validation testing and defect testing.
- Think of the system being tested **as a black box**. The system accepts inputs from some **input set I** and generates outputs in an **output set O**.
- Some of the outputs will be erroneous. These are the outputs in set $O_e$ that are generated by the system in response to inputs in the set $I_e$
- **The priority in defect testing is to find those inputs in the set $I_e$** because these reveal problems with the system.
- **Validation testing involves testing with correct inputs** that are **outside $I_e$**. These stimulate the system to generate the expected correct outputs.

# An input-output model of program testing



# Verification vs validation




✧ Verification:

"Are we building the product right".

- The software should **conform to its specification**.
- the software meets its stated **functional and non-functional requirements**

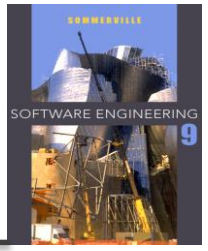
✧ Validation (more general than verification):

"Are we building the right product".

- The software should do **what the user really requires**.
- Validation is **essential** because requirements **specifications do not always reflect the real wishes or needs of system customers** and users.

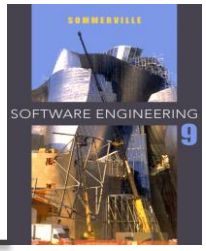
✧ These checking processes **start as soon as requirements become available and continue through** all stages of the development process.

# V & V level of confidence

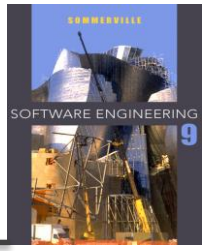



- Aim of V & V is **to establish confidence** that the system is '**fit for purpose**'.
- **Depends on system's purpose**, user expectations and marketing environment
  - Software purpose
    - The level of confidence depends on **how critical the software is** to an organisation (e.g., is **it a safety critical system**).
  - User expectations
    - Users may have low expectations of certain kinds of software.
    - Users **may tolerate failures** because the benefits of use outweigh the costs of failure recovery
  - Marketing environment
    - **Getting a product to market early** may be more important than finding defects in the program.
    - If a software product is **very cheap**, users may be willing to tolerate a lower level of reliability.

# Inspections and testing

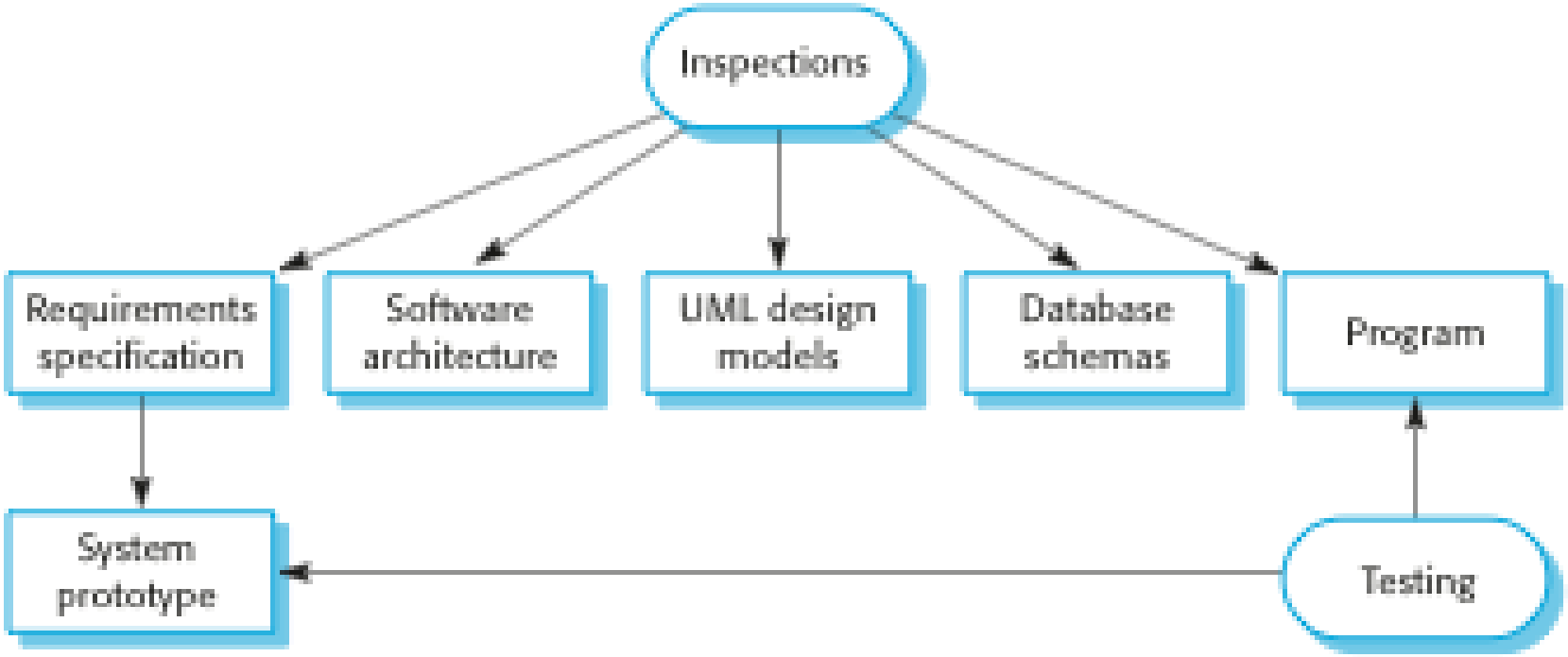



- **As well as software testing**, the verification and validation process **may involve software inspections and reviews.**
- Inspections and reviews **analyze and check**
  - the system requirements,
  - design models,
  - the program source code, and
  - even proposed system tests**.**
- These are **so-called 'static' V & V techniques** in which you **don't need to execute the software** to verify it.

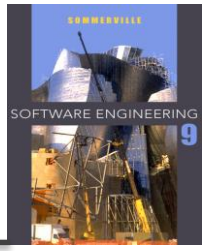



- ✧ Software inspections Concerned with analysis of the static system representation to discover problems *(***static verification**)
  - May be supplement by tool-based document and code analysis.
  - Discussed in Chapter 15.
- ✧ Software testing Concerned with exercising and observing product behaviour (**dynamic verification**)
  - The system is executed with test data and its operational behaviour is observed.

# Inspections and testing

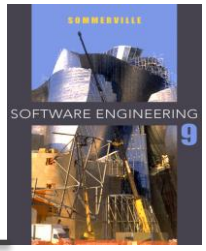


## Software inspections



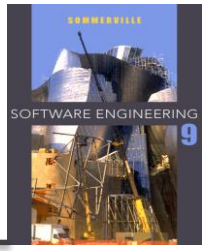

- These **involve people** examining the source representation with the aim of discovering anomalies and defects.
- Inspections **not require execution** of a system so may be **used before implementation**.
- They **may be applied to any representation of the system** (requirements, design, configuration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

## Advantages of inspections



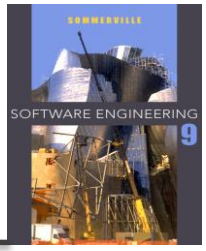

- During testing, **errors can mask (hide) other errors**. Because inspection is a static process, you don't have to be concerned with interactions between errors.
- **Incomplete versions of a system can be inspected without additional costs**. If a **program is incomplete**, then you need to develop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



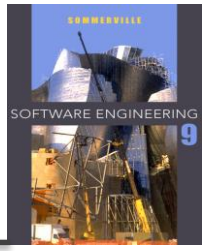

- Inspections can check conformance with a specification **but not conformance with the customer's real requirements**.
- Inspections are **not good for discovering defects** that arise because of **unexpected interactions between different parts of a program**, timing problems, or problems with system performance.
- Inspections **cannot check non-functional characteristics** such as performance, usability, etc.
- **It can be difficult and expensive** to put together a **separate inspection team** as all potential members of the team may also be software developers.

# Inspections and testing



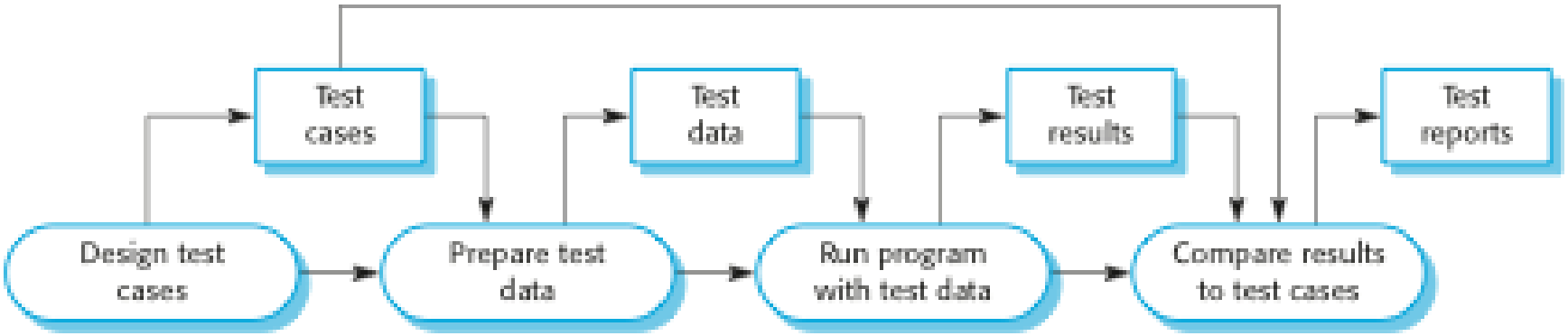

- Inspections **cannot replace** software testing
- Inspections and testing **are complementary** and not opposing verification techniques.
- **Both should be used** during the V & V process.

## Test Cases

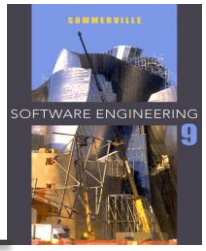



- **Test cases** are **specifications of the inputs to the test** and the **expected output** from the system (the test results),plus **a statement of what is being tested**.
- **Test data** are the inputs that have been devised to test a system.
- **Test data can sometimes be generated automatically**, **but automatic test case generation is impossible**,
  - **People** who understand what the system is supposed to do must be involved **to specify the expected test results**.
- However, **test execution can be automated**.
  - The expected **results are automatically compared with the predicted results** so there is no need for a person to look for errors and anomalies in the test run

# A model of the software testing process

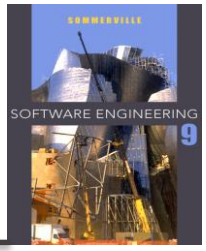


# Stages of testing



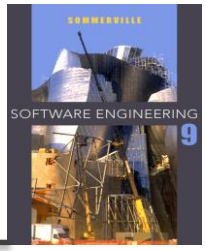

- **Development testing**, where the system is tested **during development** to discover bugs and defects.
  - **System designers and programmers** are likely to be involved in the testing process.
- **Release testing**, where a **separate testing team** test a complete version of the system before it 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.
  - **Acceptance testing is one type of user testing** where the customer formally tests a system **to decide if it should be accepted** from the system supplier or if further development is required.

# Development testing



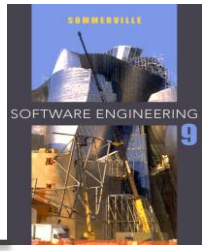

- Testing may be carried out at **three levels of granularity**:
- **Development testing includes all testing activities that are carried out by the team developing** the system.
  - **Unit testing**, where individual program units or object classes are tested. Unit testing should **focus on testing 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



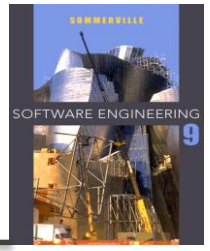

- Unit testing is the process of testing individual **components in isolation**.
- **It is a defect testing process**.
- Units may be:
  - **Individual functions** or methods within an object
  - **Object classes** with several attributes and methods
  - **Composite components** with defined interfaces used to access their functionality.

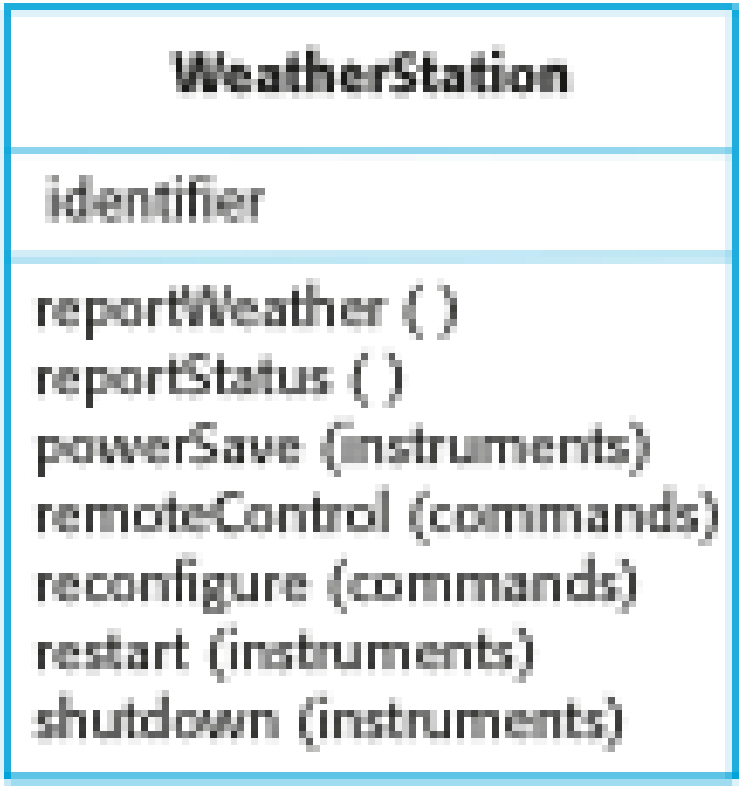
# Object class testing



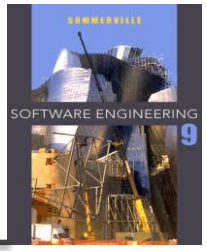

- Complete test coverage of a class involves
  - **Testing all operations** associated with an object
  - **Setting and interrogating all object attributes**
  - **Put the object into all possible states.**
    - This means that you **should simulate all events that cause a state change.**
- **Inheritance makes it more difficult to design object class tests as the information to be tested is not localised**.
- You have to **test the inherited operation in all of the contexts where it is used**.

# The weather station object interface



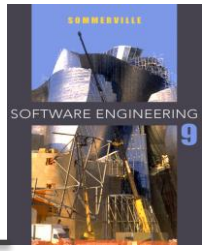



# Weather station testing



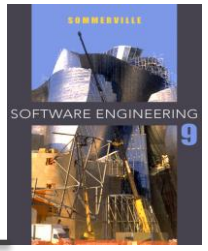

- Need to define test cases for reportWeather, calibrate, test, startup and shutdown.
- Using a state model, **identify sequences of state transitions to be tested** and the event sequences to cause these transitions
- For example:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing -> 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 of otherwise of the tests.

# Automated test components

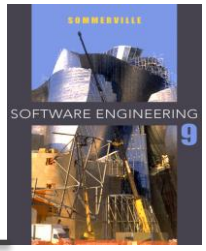



✧ **An automated test has three parts:**

1. **A setup part**, where you i**nitialize the system with the test case**, namely the inputs and expected outputs.
2. **A call part**, where **you call the object or method to be tested.**
3. **An assertion** part where **you compare the result of the call with the expected resu**lt.

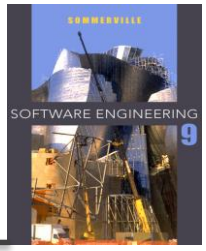
If the **assertion evaluates to true**, **the test has been successful if false, then it has failed**.

# Using Mock Objects



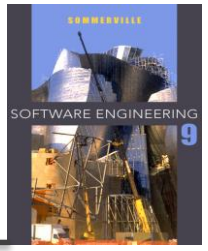

- Sometimes the object that you are testing has **dependencies on other objects** that may not have been written or which slow down the testing process if they are used.
  - For example, **if your object calls a database**, this may involve **a slow setup process** before it can be used.
- In these cases, you may decide to **use mock objects**. Mock objects are **objects with the same interface as the external objects** being used that simulate its functionality.
- **A mock object simulating a database** may have only a **few data items that are organized in an array**.

## Choosing unit test cases:




- Testing is **expensive and time consuming**, so it is important that **you choose effective 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.**

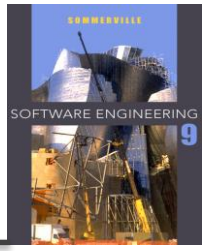
## 2 types of unit test case




✧This leads **to 2 types of unit test case**:

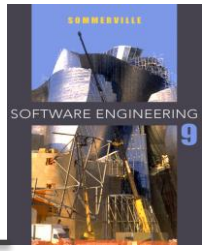
- The first of these should **reflect normal operation o**f a program and should show that the component works as expected.
- The other kind of test case should be based on testing experience of where common problems arise. It should use **abnormal inputs** to check that these are properly processed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

## write two kinds of test case

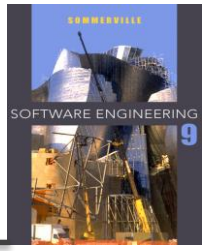



- You should therefore write two kinds of test case. The **first of these should reflect normal operation** of a program and should show that the component works.
  - For example, if you are testing a component that creates and initializes a new patient record, then your test case should show that the record exists in a database and that its fields have been set as specified.
- 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

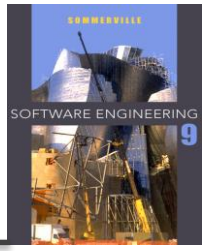



1. **Partition testing**, where you identify groups of inputs that have common characteristics and should be processed in the same way.
   - Examples of these classes are **positive numbers, negative numbers, and menu selections**
   - That is if you test a program that does a computation and requires two positive numbers, then you would expect the program
     - to behave in the same way for all positive numbers.
     - Because of this equivalent behavior, these classes are **sometimes called equivalence partitions or domains**
     - Test cases are designed so that the **inputs or outputs lie within these partitions**

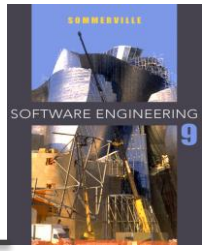



2. **Guideline-based testing**, where you use testing guidelines to choose test cases.
   - These guidelines **reflect previous experience** of the kinds of errors that programmers often make when developing components.

# Partition testing

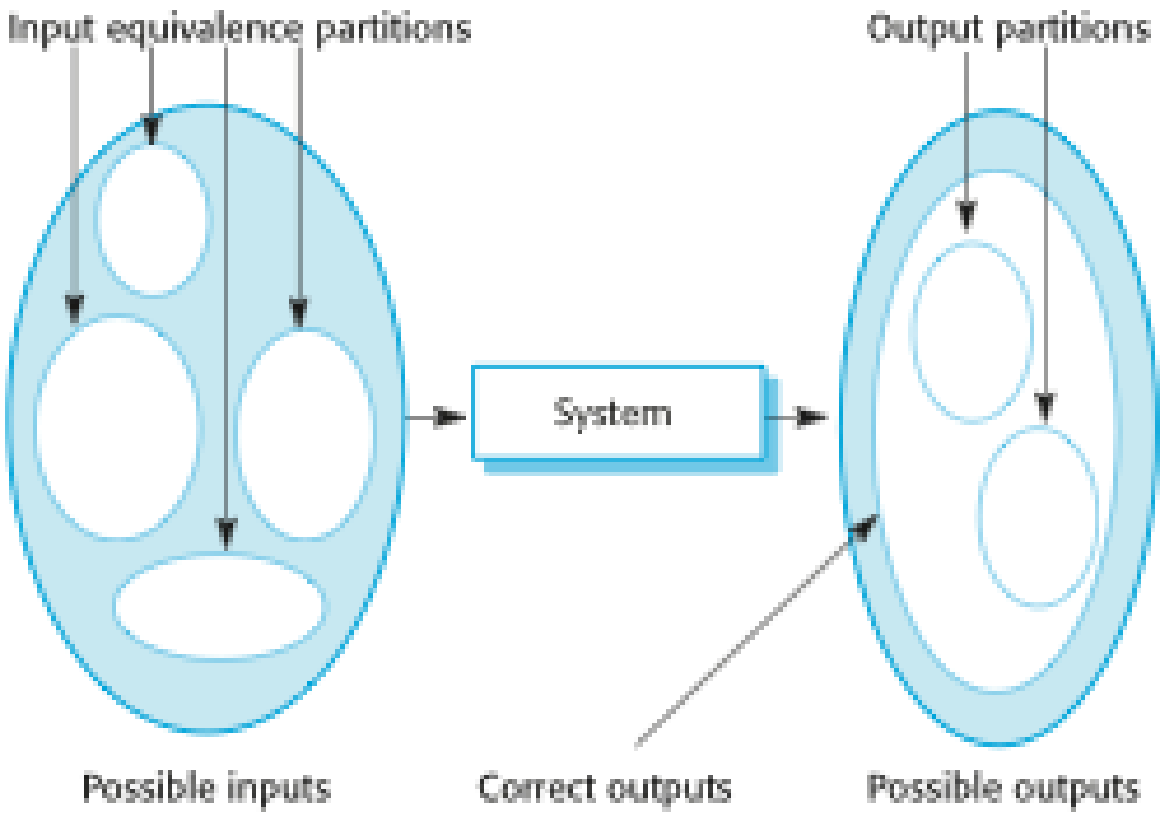



- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

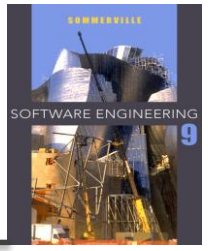



- In the following Figure, the **large shaded ellipse** on the left represents the **set of all possible inputs** to the program that is being tested.
- The **smaller unshaded ellipses represent equivalence partitions**.
- **A program being tested should process all of the members of an input equivalence partitions in the same way.**
- **Output equivalence partitions** are partitions within which all of the outputs **have something in common**.
- **Sometimes there is a 1:1 mapping** between input and output equivalence partitions.
- However, this is **not always the case**; you may need to define a separate input equivalence partition, where the only common characteristic of the inputs is that they generate outputs within the same output partition.
- The **shaded area in the left ellipse represents inputs that are invalid**.
- The **shaded area in the right ellipse represents exceptions** that may occur (i.e., responses to invalid inputs).

# Equivalence partitioning

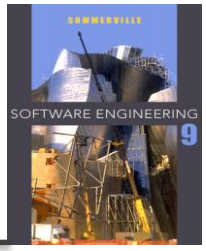


## A good rule of thumb



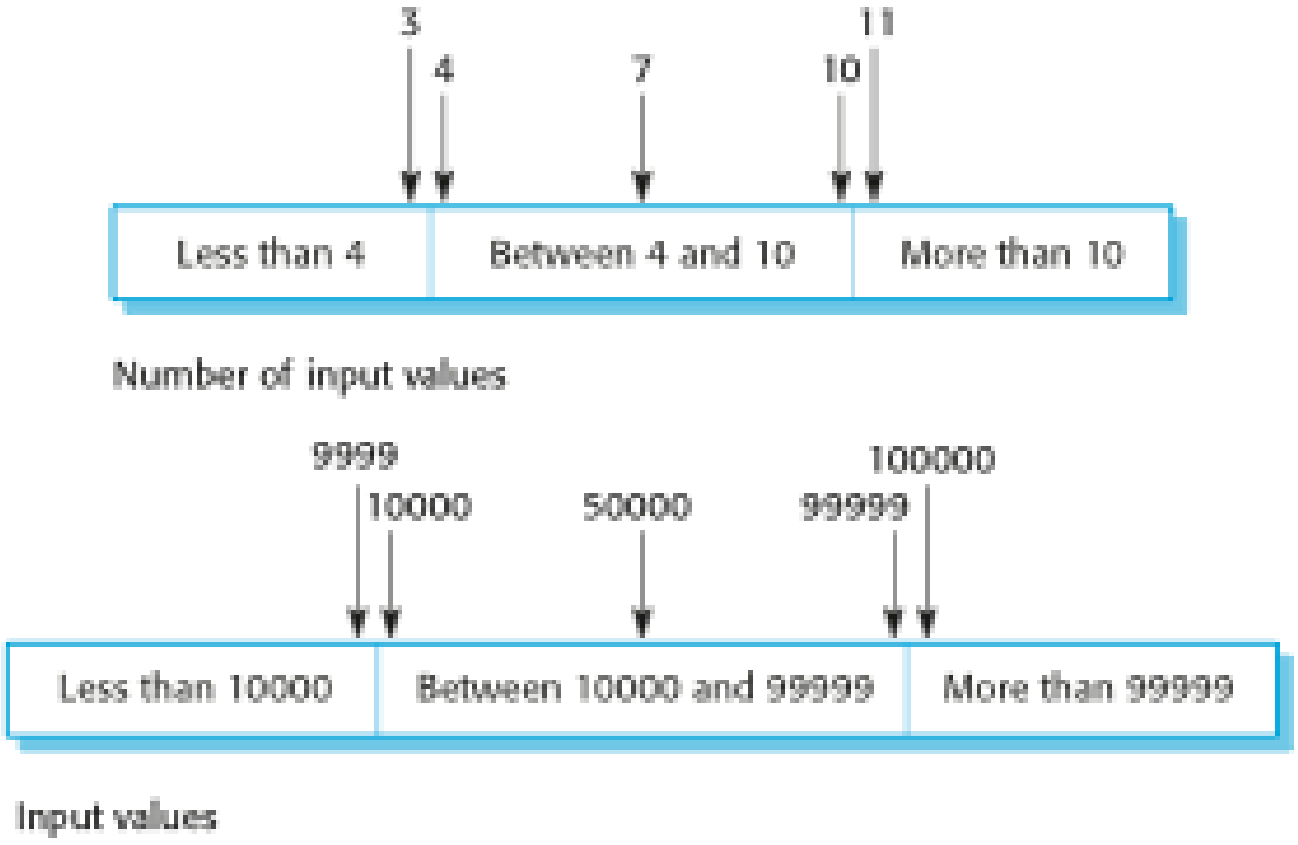

- Choose test cases **on the boundaries of the partitions**, **plus cases close to the midpoint** of the partition.
- The reason for this is that designers and **programmers tend to consider typical values** of inputs when developing a system. You test these by choosing the midpoint of the partition.
- **Boundary values** are often atypical (e.g., zero may **behave differently** from other non-negative numbers) so are sometimes overlooked by developers.
- **Program failures often occur when processing these atypical values.**

# Example

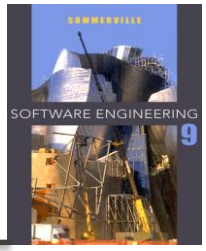



- **You identify partitions** by using **the program specification or user documentation** and **from experience** where you predict the classes of input value that are likely to detect errors.
- For example, say a program specification states that **the program accepts 4 to 10 inputs which are five-digit integers greater than 10,000**.
- You use this information to identify the input partitions and possible test input values

# Equivalence partitions

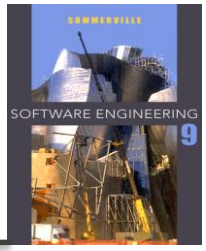


# Black-box vs White Box Testing




- **When you use the specification of a system** to identify equivalence partitions, this is called 'black-box testing'. Here, **you don't need any knowledge of how the system works**.
- However, **it may be helpful to supplement the black-box tests with 'white-box testing'**, **where you look at the code of the program to find other possible tests**.
- For example, **your code may include exceptions** to handle incorrect inputs.
- **You can use this knowledge to identify 'exception partitions'**—different ranges where the same exception handling should be applied.

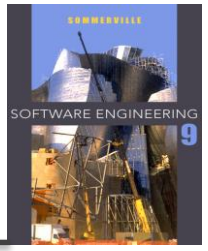
# 2. Testing guidelines (sequences)




✧ For example, when you are testing programs with sequences, arrays, or lists, guidelines that could help reveal defects include:

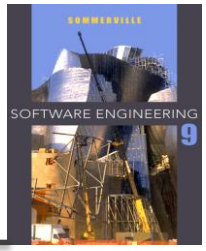
- Test software with sequences which have only a **single value**.
  - If presented with a single-value sequence, a program may not work properly.
- Use **sequences of different sizes** in different tests.
- Derive tests so that **the first, middle and last elements of the sequence are accessed**.
- Test with **sequences of zero length**.

## General testing guidelines



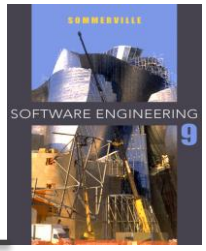

- Whittaker's book (2002) **includes many examples of guidelines that can be used in test case design**. Some of the most general guidelines that he suggests are:
  - Choose inputs that **force the system to generate all error messages**
  - Design inputs that **cause input buffers to overflow**
  - **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**.

# Path testing

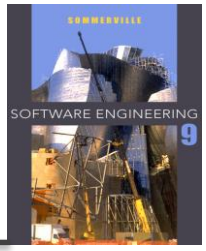



- Path testing is a testing strategy that **aims to exercise every independent execution path** through a component or program.
- If every independent path is executed, then all **statements** in the **component must have been executed at least once.**
- All **conditional statements** are tested for both true and false cases.
- In an **object-oriented** development process, path testing may be used **when testing the methods associated with objects.**

## Key points



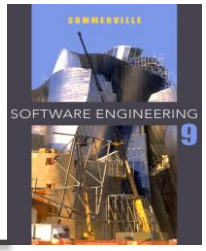

- Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.
- Development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



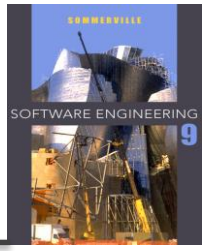

# Chapter 8 – Software Testing

Lecture 2

# Component testing

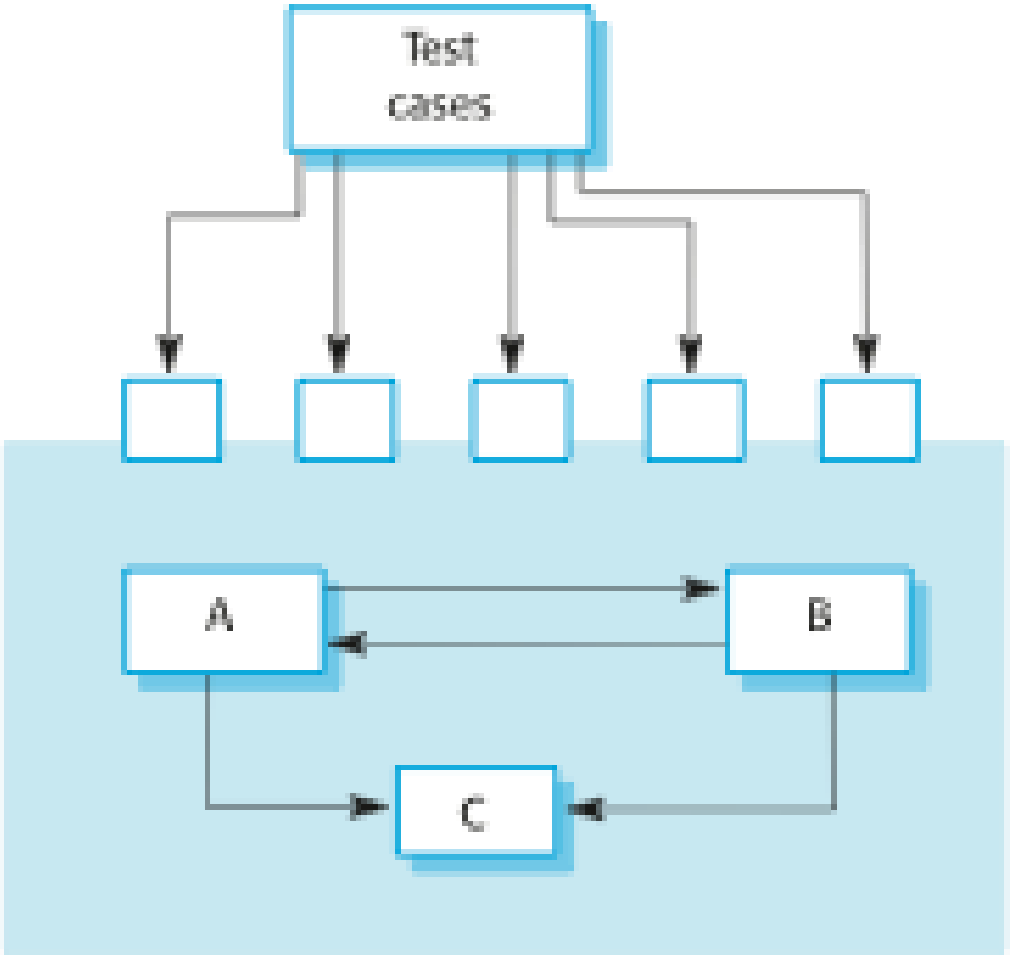



- Software components are often composite components that are **made up of several interacting objects**.
  - For example, in the weather station system, the reconfiguration component **includes objects that deal with each aspect of the reconfiguration.**
- You access the f**unctionality** of these objects **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**.

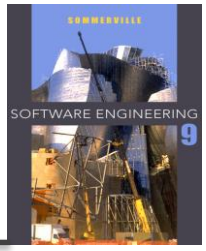



- **The following figure illustrates the idea of component interface testing**.
- Assume that **components A, B, and** C have been integrated to create a larger component or subsystem.
- The **test cases are not applied to the individual components but rather to the interface of the composite component** created by combining these components.
- Interface errors in the composite component **may not be detectable by testing the individual objects**
- because **these errors result from interactions between the objects in the component**.

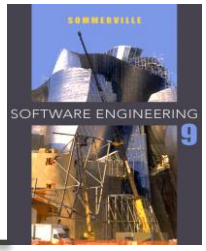
# Interface testing



# Interface testing

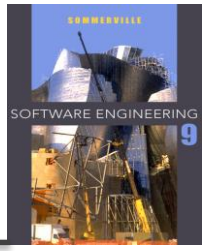



- Objectives are to **detect faults due to interface errors** or **invalid assumptions** about interfaces.
- Interface types
  - Parameter interfaces Data passed from one method or procedure to another. Methods in an object have a parameter interface.
  - Shared memory interfaces Block of memory is shared between procedures or functions. Data is placed in the memory by one subsystem and retrieved from there by other sub-systems. This type of interface is often **used in embedded systems**, where sensors create data that is retrieved and processed by other system components.




- Procedural interfaces Sub-system encapsulates a set of procedures to be called by other sub-systems.
  - Objects and reusable components have this form of interface.
- Message passing interfaces Sub-systems request services from other sub-systems by passing a message to it.
  - A **return message includes the results** of executing the service.
  - Some **object-oriented systems** have this form of interface, as do **client–server systems**.

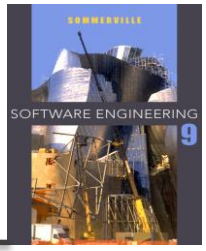
# Interface errors




✧ Interface errors are **one of the most common forms of error** in complex systems (Lutz, 1993).

✧ These errors fall into three classes:

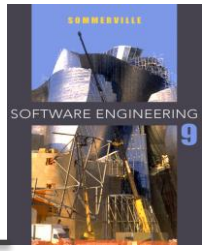
1. **Interface misuse**
   - A calling component calls another component and makes an error in its use of its interface **e.g. parameters in the wrong order, number, or type.**
2. **Interface misunderstanding**
   - A calling component **embeds assumptions** about the behaviour of the called component **which are incorrect**.
   - For example, **a binary search method** may be called with a parameter that is an unordered array. The search would then fail.




## 3. Timing errors

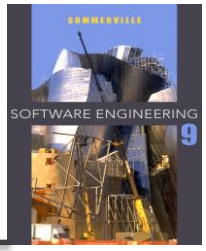
- The called and the calling component **operate at different speeds** and out-of-date information is accessed.
- For example in real time systems **the producer of data and the consumer of data may operate at different speeds**.
- Unless particular care is taken in the interface design, the **consumer can access out-of-date information**

## Interface testing guidelines

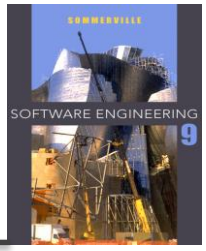



- Design tests so that parameters to a called procedure are **at the extreme ends of their ranges**.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- **Use stress testing** in message passing systems.
  - This means that you should **design tests that generate many more messages** than are likely to occur in practice.
  - This is an effective way of revealing timing problems.
- **In shared memory systems**, **vary the order in which components are activated.**
  - These tests may **reveal implicit assumptions** made by the programmer **about the order in which the shared data is produced and consumed.**

## System testing

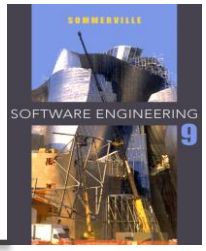



- ✧ System testing during development **involves integrating components to create a version of the system** and then **testing the integrated system**.
- ✧ The focus in system testing is **testing the interactions between components.**
- ✧ System testing **checks that**
  - components are compatible,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



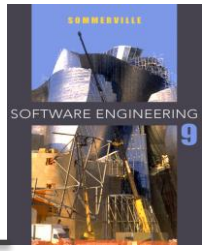

- When you integrate components to create a system, you get emergent behavior.
- This means that some elements of system **functionality only become obvious when you put the components together.**
- This may be planned emergent behavior, which has to be tested.
- For example, you may integrate an authentication component with a component that updates information.
- You then have a system **feature that restricts information updating to authorized users**.
- Sometimes, however, **the emergent behavior is unplanned and unwanted.**
- You have to develop tests that **check that the system is only doing what it is supposed to do.**

# System and component testing



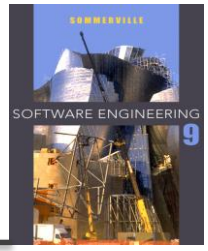

- During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. **The complete system is then tested**.
- Components **developed by different team members** or sub-teams may be integrated at this stage. System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

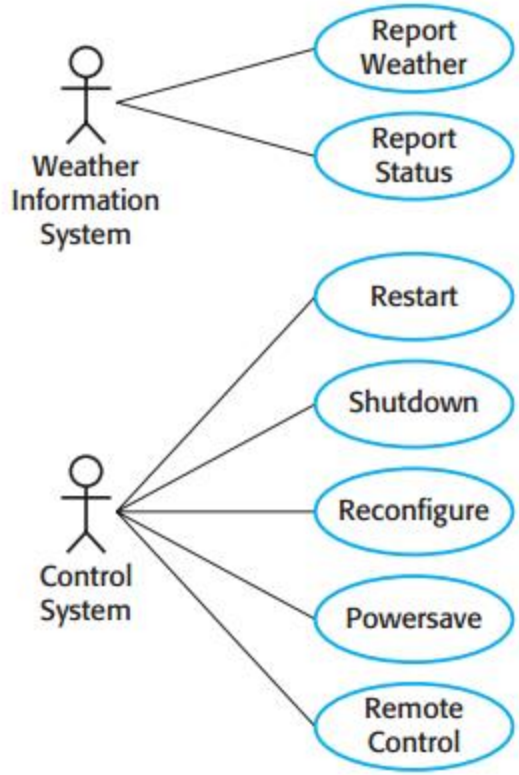
## Use-case testing



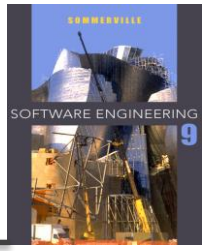

- The **use-cases** developed to identify system interactions can be **used as a basis for system testing**.
- Each use case **usually involves several system components** so testing the use case **forces these interactions to occur.**
- **The sequence diagrams** associated with the use case **documents the components and interactions that are being tested.**
- The sequence diagram **helps you design the specific test cases** that you need as it shows **what inputs are required and what outputs are created**:

# Weather Station Use Cases



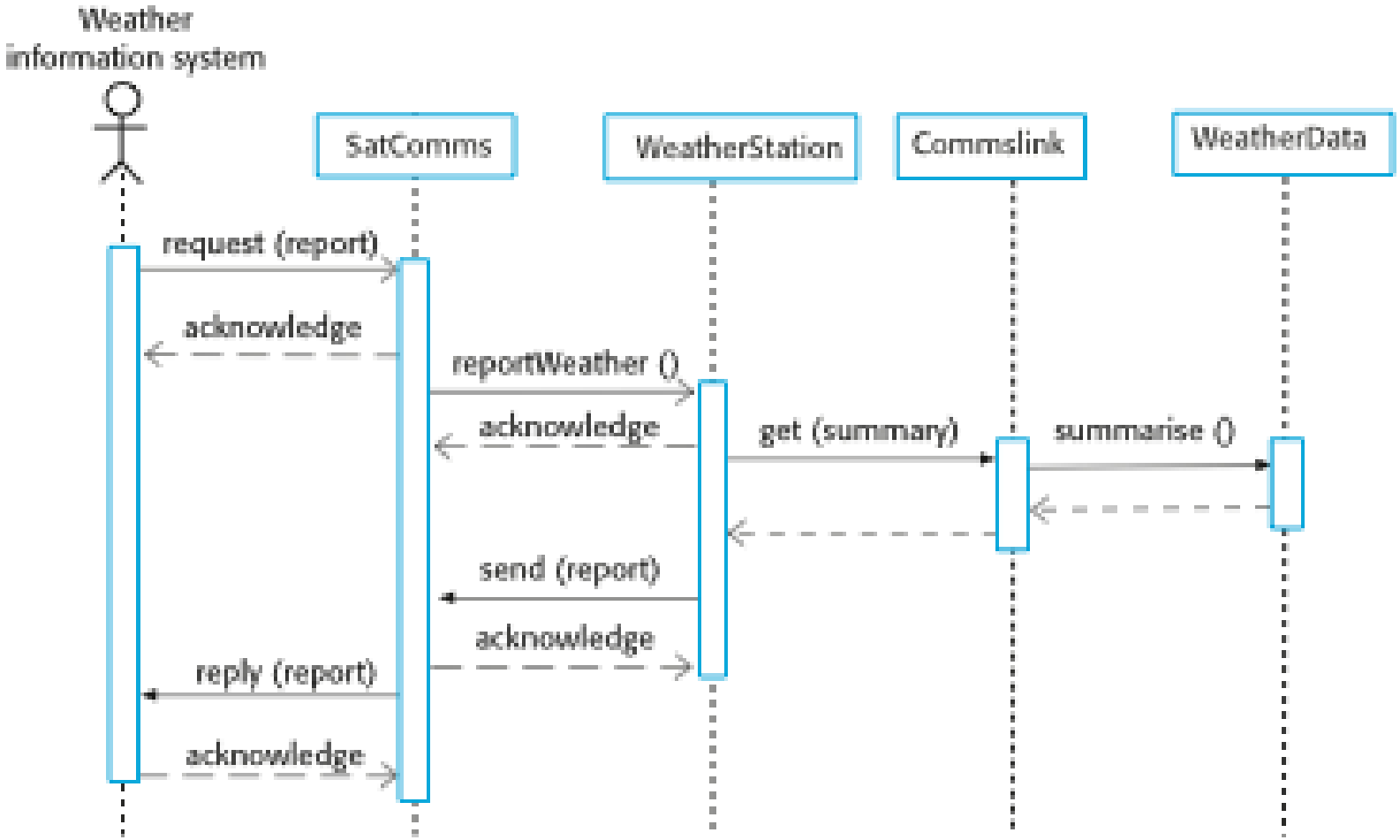



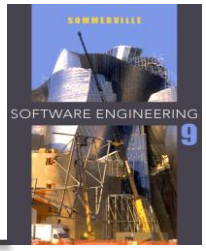
# Use Case Description-Report Weather




| | |
|---|---|
| **System** | Weather station |
| **Use case** | Report weather |
| **Actors** | Weather information system, Weather station |
| **Dat** | The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals. |
| **Stimulus** | The weather information system establishes a satellite communication link with the weather station and requests transmission of the data. |
| **Response** | The summarized data are sent to the weather information system. |
| **Comments** | Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future. |

# Collect weather data sequence chart



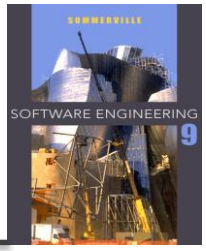



- You can use this diagram to identify operations that will be tested and to help design the test cases to execute the tests.
- Therefore, issuing a request for a report will result in the execution of the following thread of methods:

**SatComms:request → WeatherStation:reportWeather → Commslink:Get(summary) → WeatherData:summarize**

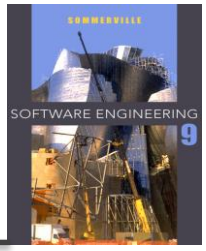
- The sequence diagram helps you design the specific test cases that you need as it shows what inputs are required and what outputs arecreated:

# Testing policies



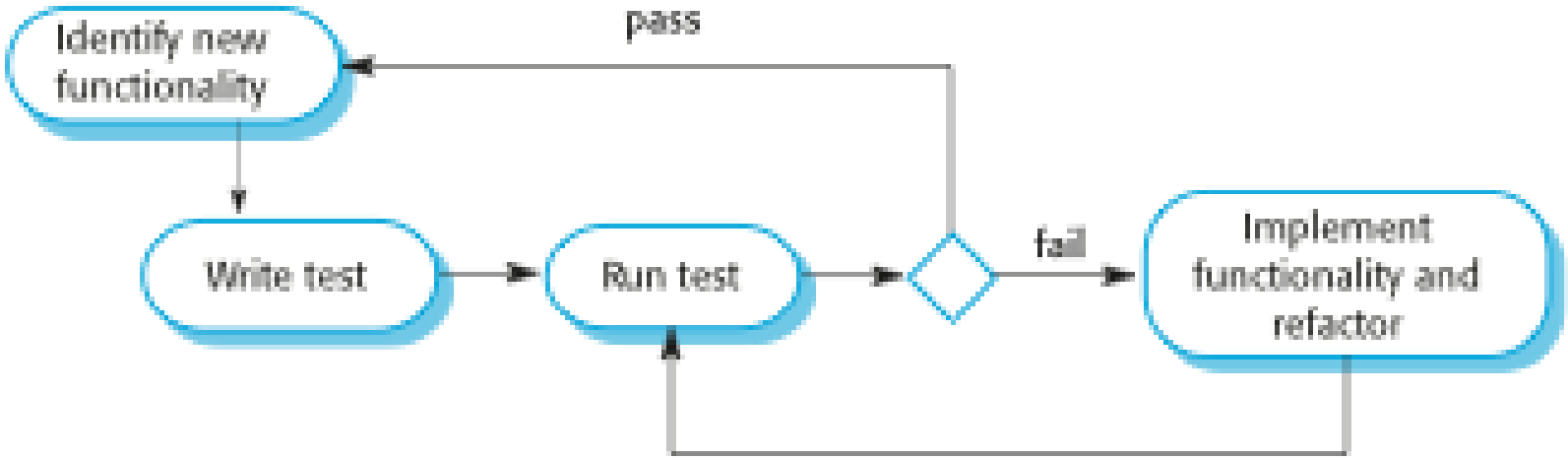

- **Exhaustive system testing** is impossible so testing policies which define the required system test coverage may be developed.
- Examples of testing policies:
  - All system functions that are accessed through menus should be tested.
  - Combinations of functions (e.g. text formatting) that are accessed 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

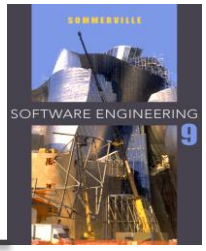



- Test-driven development (TDD) is **an approach to program development** in which **you inter-leave testing and code development.**
- **Tests are written before code** and '**passing' the tests is the critical driver** of development.
- **You develop code incrementally**, along with a test for that 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

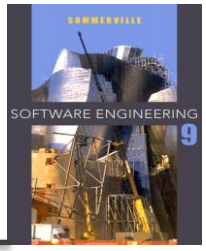


## TDD process activities



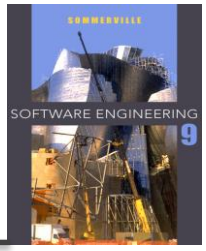

- **Start by identifying the increment of functionality** that is required. This should normally be small and implementable in a few lines of code.
- **Write a test** for this functionality and **implement this as an automated test**.
- **Run the test**, **along with all other tests** that have been implemented. Initially, you have **not implemented** the functionality **so the new test will f**ail.
- **Implement the functionality and re-run the test**. This **may involve refactoring** existing code to improve it and add new code to what's already there.
- **Once all tests run successfully**, you move on to implementing the next chunk of functionality.

## Benefits of test-driven development



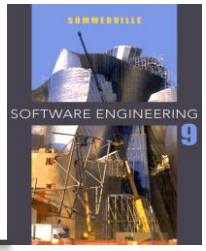

- A strong argument for test-driven development is that it **helps programmers clarify their ideas** of what a code segment is actually supposed to do.
- To write a test, **you need to understand what is intended**, as this understanding makes it easier to write the required code.
- Of course, **if you have incomplete knowledge** or understanding, then test-driven development won't help.
- If you don't know enough to write the tests, you won't develop the required code.

# Benefits of test-driven development



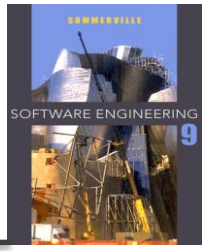

- ✧ Code coverage
  - Every code segment that you write has at least one associated test so all code written has at least one test.
- ✧ Regression testing
  - A regression test suite is developed incrementally as a program is developed.
- ✧ Simplified debugging
  - When a test fails, it should be obvious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



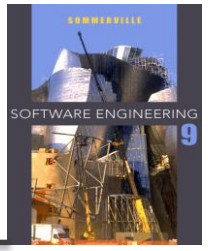

- Regression testing is testing the system to check that **changes have not 'broken' previously working c**ode.
- In a manual testing process, **regression testing is expensive** but, with **automated testing**, it is simple and straightforward.
- **All tests are rerun every time a change is made to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

## Release testing



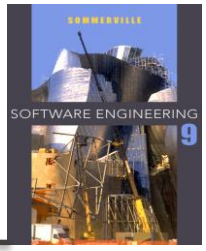

- Release testing is the process of **testing a particular release of a system** that is **intended for use outside** of the development team.
- **The primary goal** of the release testing process is to convince the supplier of the system that **it is good enough for use**.
  - Release testing, therefore, 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



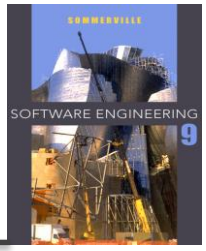

- Release testing is **a form of system testing**.
- Important differences:
  - **A separate team** that has not been involved in the system development, should be **responsible for release testing**.
  - System testing by the development team should focus on discovering bugs in the system (defect testing).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

## Requirements based testing



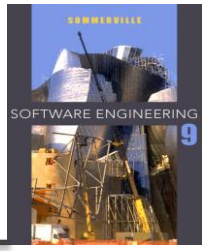

- Part of release testing
- A general principle of good requirements engineering practice is that **requirements should be testable**; that is, **the requirement should be written** so that a test can be designed for that requirement.
- **A tester can then check** that the requirement has been satisfied.
- **You consider each requirement and derive a set of tests for it.**
- Requirements-based testing **is validation rather than defect testing**—you are trying to

## Requirements based testing



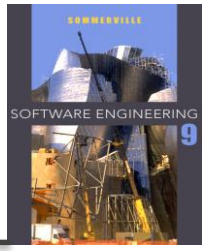

- Requirements-based testing **involves examining each requirement** and developing a test or tests for it.
- MHC-PMS requirements:
  - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
  - If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.

# Requirements tests



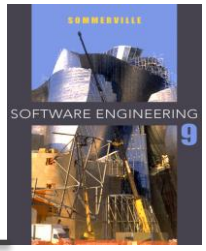

- **Set up a patient record with no known allergies**. Prescribe medication for allergies that are known to exist. **Check that a warning message is not issued by the system.**
- Set up a patient record **with a known allergy**. Prescribe the medication to that the patient is allergic to, and **check 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 c**heck 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.**

## Scenario testing



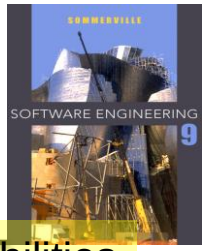

- Scenario testing is **an approach to release testing** where you **devise typical scenarios of use** and use these to develop test cases for the system.
- A scenario is a **story that describes one way in which the system might be used**.
- Scenarios **should be**
  - **realistic** and
  - **real system users should be able to relate to them**.

## 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.

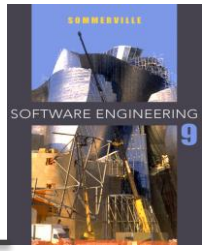
# A usage scenario for the MHC-PMS




Kate is a nurse who specializes in mental health care. One of her 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, Kate logs into the MHC-PMS and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

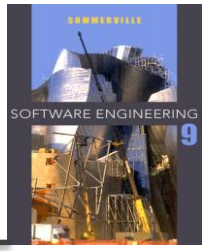
One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side -effect of keeping him awake at night. Kate looks up Jim's record and is prompted for her key phrase to decrypt the record. She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

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



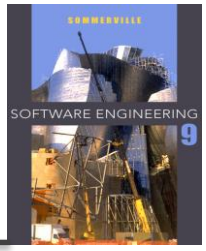

- If you are a release tester, **you run through this scenario**, **playing the role of Kate** and observing how the system behaves in response to different inputs.
- As ‘Kate’, you may **make deliberate mistakes**, such as inputting the wrong key phrase to decode records. This checks the response of the system to errors.
- You should **carefully note any problems** that arise, including **performance problems**. If a system is too slow, this will change the way that it is used.
- For example, **if it takes too long to encrypt** a record, then users who are short of time **may skip this stage**.
- **If they then lose their laptop**, an unauthorized person could then view the patient records.

# Performance testing



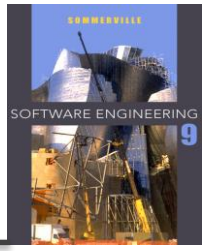

- **Part of release testing** may involve testing the **emergent properties** of a system, **such as performance and reliability**.
- Tests should **reflect the profile of use of the system**.
- Performance tests usually **involve planning a series of tests** where the load is steadily increased until 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



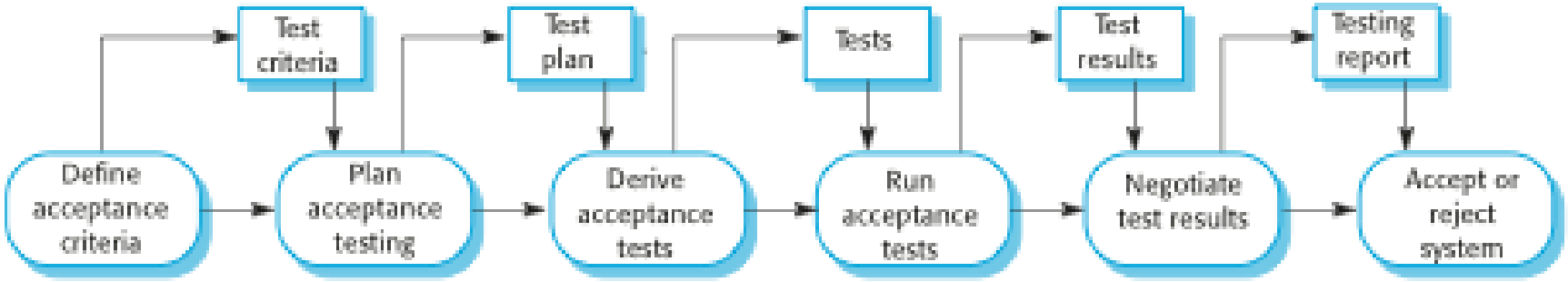

- User or customer testing is **a stage in the testing process** in which users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working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

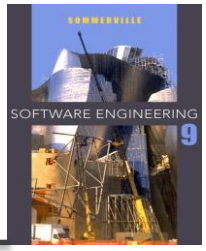



- ✧ Alpha testing
  - Users of the software work with the development team to test the software **at the developer's site**.
- ✧ Beta testing
  - A release of the **software is made available to users** to allow them to experiment and to raise problems that they discover with the system developers.
-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

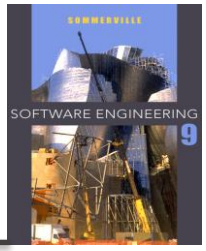


# Stages in the acceptance testing process



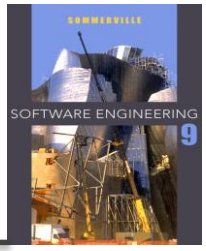

- ✧ Define acceptance criteria
- ✧ Plan acceptance testing
- ✧ Derive acceptance tests
- ✧ Run acceptance tests
- ✧ Negotiate test results
- ✧ Reject/accept system

# 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**.

# 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.