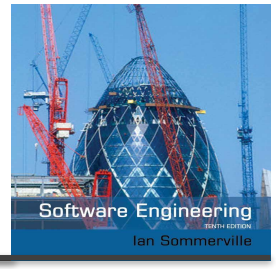
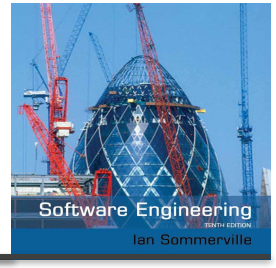
Chapter 8: software Testing

Component testing



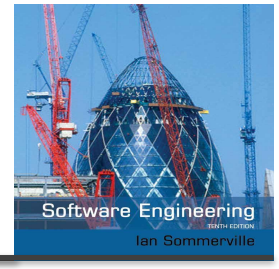
- Software components are often composite components that are made up of several interacting objects.
  - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- the **functionality of these objects is accessed via the defined component interface.**
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - Assuming that unit testing of each functionality is done already

Interface testing



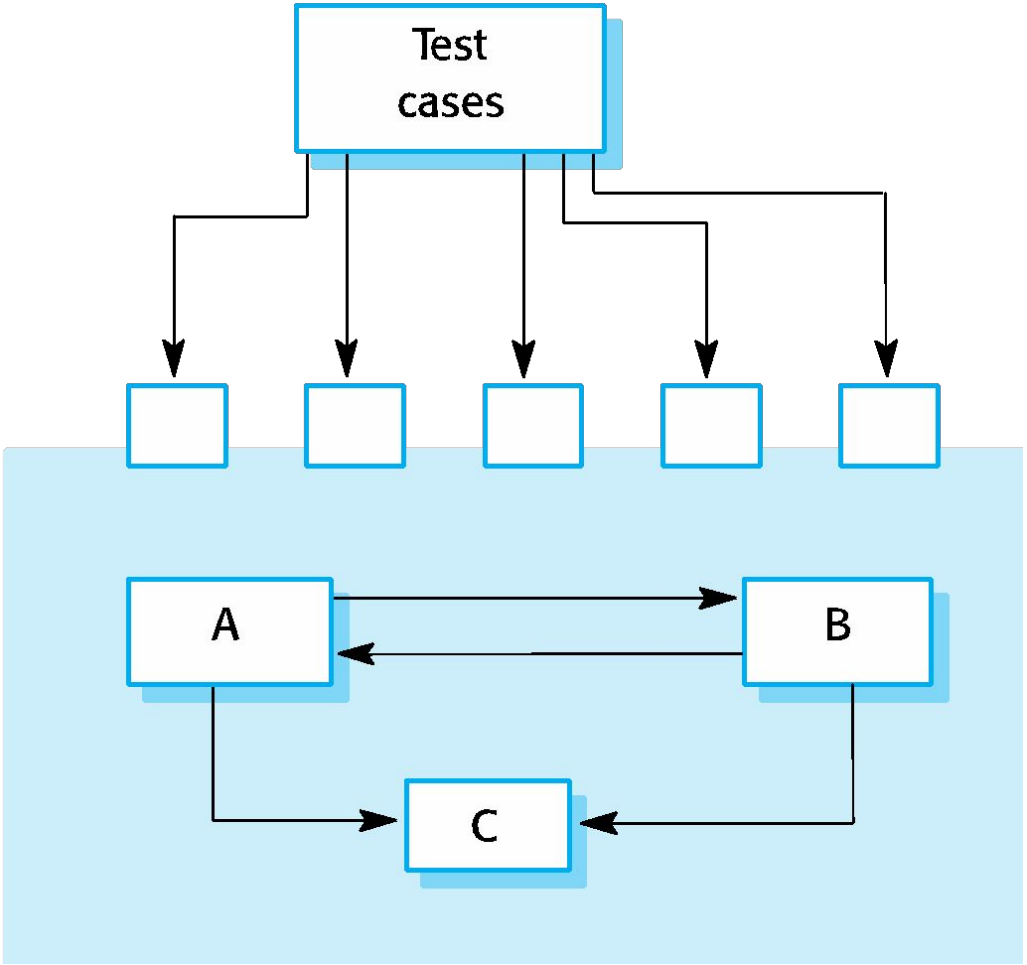
- Testing the components that helps the interaction between different components.
- Interface errors can't be identified from unit testing
- As they because of interaction between two units.

Interface testing

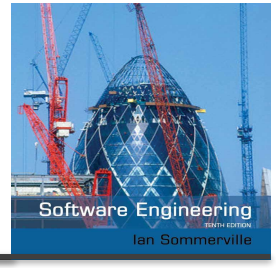


- Interface types between program components:
  - Parameter interfaces
    - Data passed from one method or procedure to another.
    - E.g., methods in object have a parameter interface, (passing parameters to multiple methods)
  - Shared memory interfaces
    - Same Block of memory is shared between different components.
    - Like in embedded systems, sensor component store value in db and analyser component may work on these values.
  - 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
    - E.g., MPI in client server arch.

Interface testing

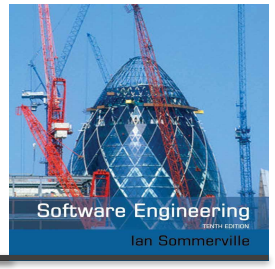


Types of Interface errors



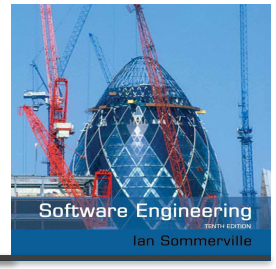
- Interface misuse
  - A calling component calls another component and makes an error in its use of its interface **e.g. parameters in the wrong order.**
- Interface misunderstanding
  - A **calling component embeds assumptions about the behaviour of the called component which are incorrect.**
  - **E.g., a binary search method is called using an unsorted array passed as a parameter.**
- Timing errors
  - In real time systems using shared memory in which the calling function access the outdated information and the saving function hasn't uploaded the latest value yet.
  - Because they are operating at different speeds.

Interface testing challenges via examples



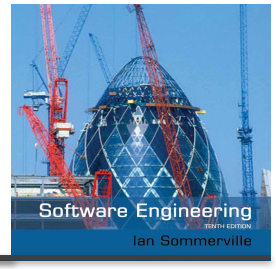
- Testing for errors in interface is difficult.
- For example, an object is accepting a queue as a finite data structure
- But the calling object assumes that the queue is infinite and it doesn't check for overflow conditions while entering an element.
- Another difficulty is that the fault propagation is detected only when multiple components are tested together, as **the faulty component will pass wrong value to the calling component** and hence the error propagates.
  - So wrong value processing in 1 component leads to an error in another component these are the interface errors.

Interface testing guidelines



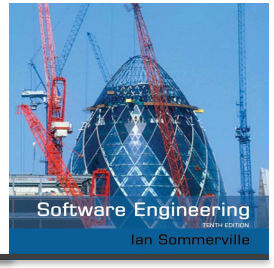
- Design tests so that parameters to a **called procedure are at the extreme ends of their ranges**. E.g., Checking for overflow and underflow conditions in an array.
- **Design tests which cause the component to fail.**
- **Use stress testing** in message passing systems to **check for timing errors.**

System testing



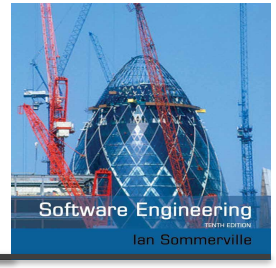
- **integrate components to create a version of the system and then testing the integrated system.**
- The focus in system testing is **testing the interactions between components.**
- System testing **checks that components are compatible**, **interact correctly** and **transfer the right data at the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

Difference between System and component testing



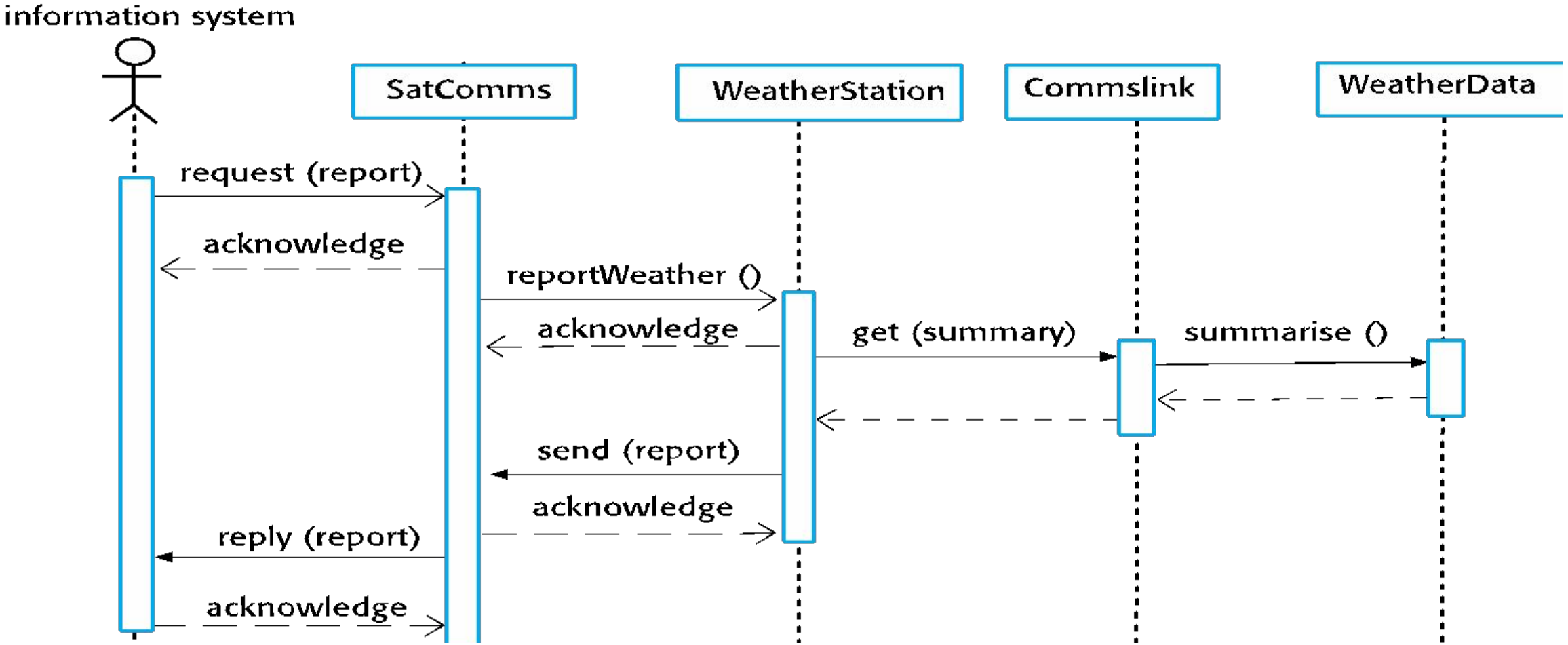
- During system testing, **reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components.** The complete system is then tested. Also, SIT (System & integration testing)
- **Components developed by different team members or sub-teams 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 based system testing

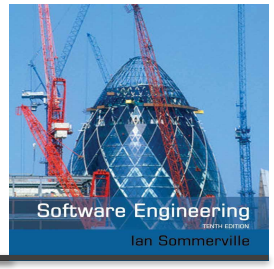


- Since system testing is testing for interactions, so its good to create use cases for that.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- The **sequence diagrams** associated with the use case documents **shows the components and interactions that are being tested.**

Collect weather data sequence chart

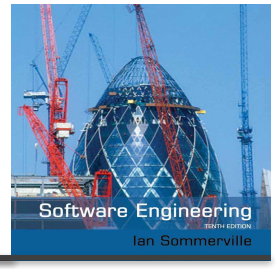


Test cases derived from sequence diagram

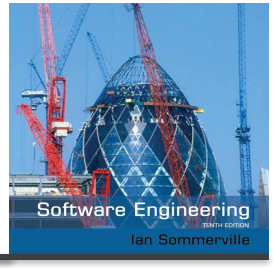


- **An input of a request for a report should have an associated acknowledgement**. A report should ultimately be returned from the request.
  - You should create summarized data that can be used to check that the report is correctly organized.
- **An input request for a report to WeatherStation results 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

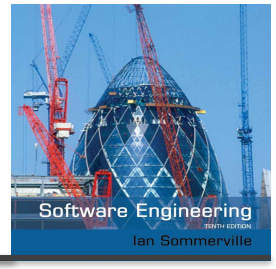


- Exhaustive system testing is impossible
- so testing policies should be defined mostly guidelines based or depends experience of system usage.
- Examples of testing policies:
  - All system functions that are accessed through menus should be tested.
  - Combinations of functions (e.g. text formatting) 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.**



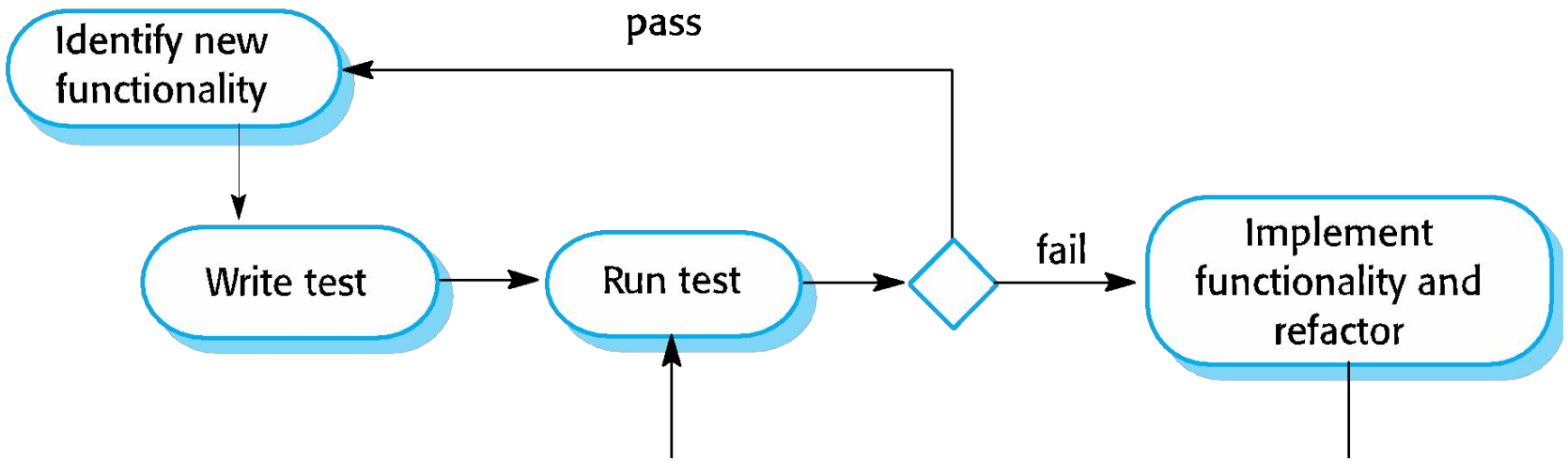
Test-driven development

Test-driven development

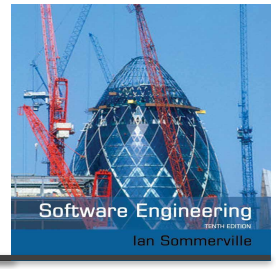


- Test-driven development (TDD) is an **approach to program development** in which you **inter-leave testing and code development**.
- **Tests are written before code**
- and '**passing' the tests is the critical driver of development.**
- You **develop code incrementally**, along with a test for that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

Test-driven development

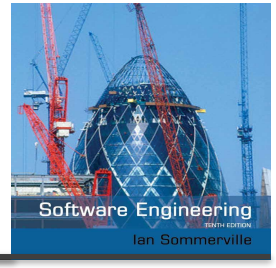


TDD process activities explained



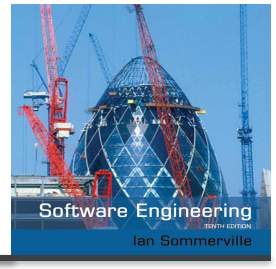
- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, 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. Defects discovered early in development stage.
- Regression testing
  - Run a regression test to check if new code added issues to previous one.
- Simplified debugging
  - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified. No need for debugging tools.
- System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing.

Regression testing



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