Incrementalism in Software Development

SWEPM Unit 3.1

Introduction to Incrementalism

- **Definition:** Incrementalism is the practice of delivering software in small, functional increments rather than as a monolithic release.
- **Why It Matters:**
  - Enables feedback loops.
  - Reduces risks and improves quality.
  - Supports adaptability in a rapidly changing environment.

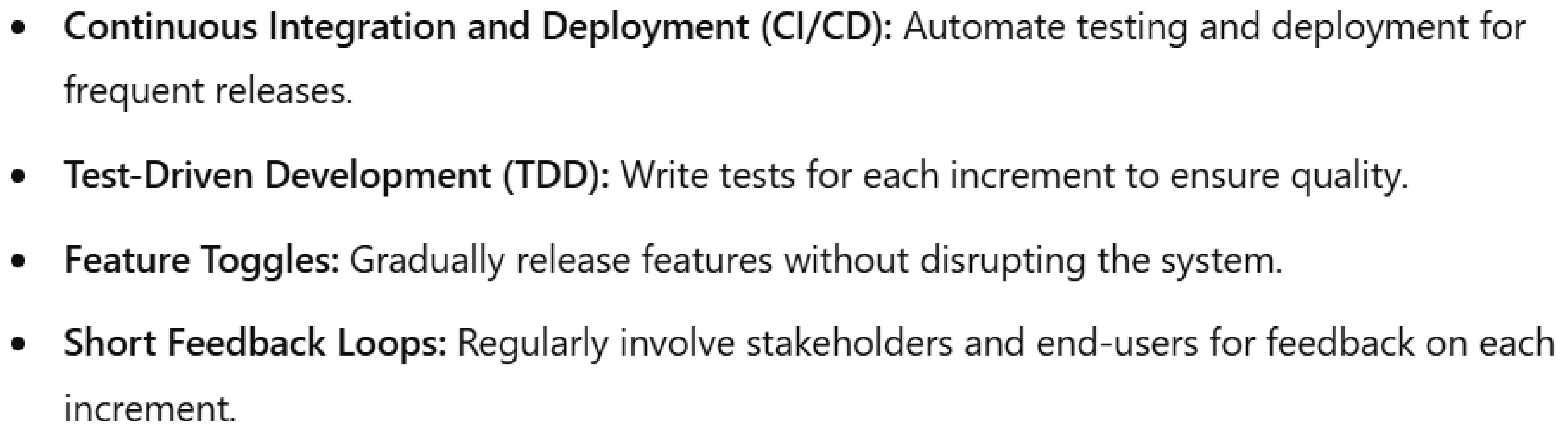
Key Principles of Incremental Development

- **Start Small, Aim Big:** Deliver small pieces of functionality with a vision for the whole system.
- **Frequent Deliveries:** Regularly release increments that deliver real value.
- **Feedback-Driven Development:** Use feedback from each increment to refine future plans and designs.
- **Quality at Every Step:** Maintain high standards of quality for every increment.

Benefits of Incremental Development

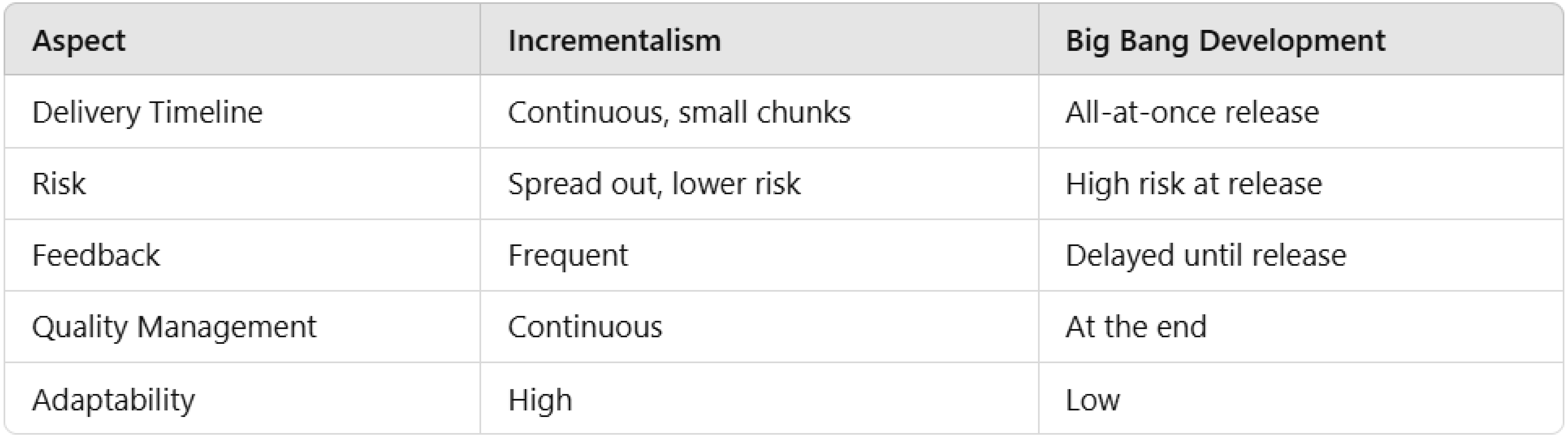
- **Reduced Risk:**
  - Smaller increments mean fewer chances of catastrophic failure.
- **Faster Feedback:**
  - Immediate feedback helps guide the project in the right direction.
- **Improved Collaboration:**
  - Frequent deliveries foster better communication among stakeholders.
- **Easier to Manage Complexity:**
  - Building incrementally allows teams to focus on smaller, manageable parts of the problem.

Core Practices Supporting Incrementalism



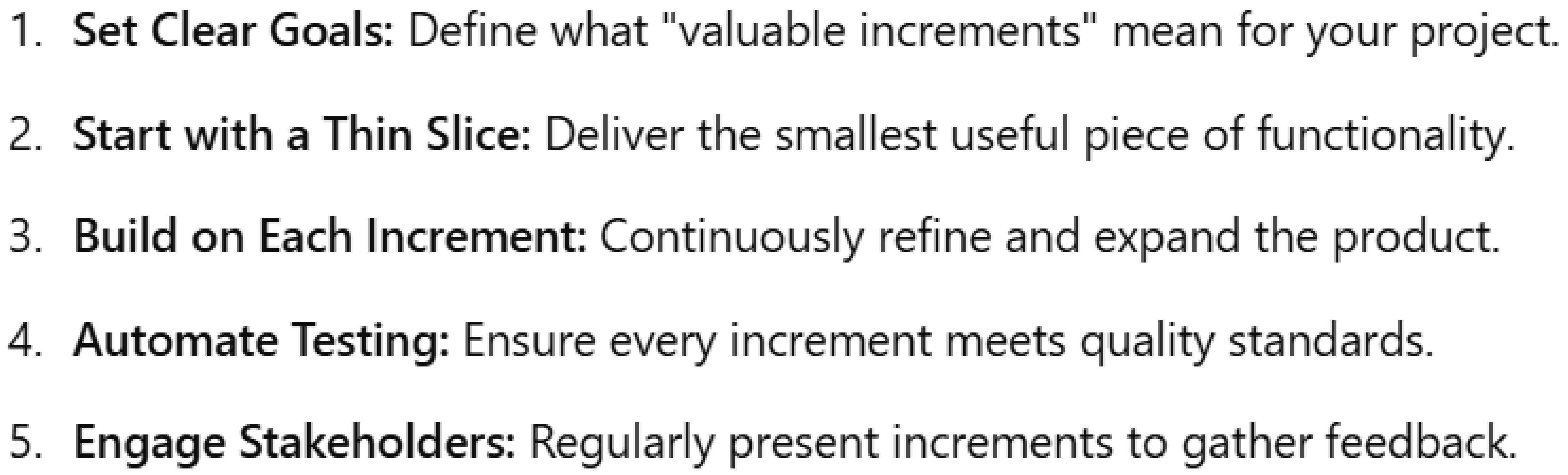
- **Continuous Integration and Deployment (CI/CD):** Automate testing and deployment for frequent releases.
- **Test-Driven Development (TDD):** Write tests for each increment to ensure quality.
- **Feature Toggles:** Gradually release features without disrupting the system.
- **Short Feedback Loops:** Regularly involve stakeholders and end-users for feedback on each increment.

Incrementalism vs Big Bang Development



Aspect	Incrementalism	Big Bang Development
Delivery Timeline	Continuous, small chunks	All-at-once release
Risk	Spread out, lower risk	High risk at release
Feedback	Frequent	Delayed until release
Quality Management	Continuous	At the end
Adaptability	High	Low

Steps to Adopt Incremental Development

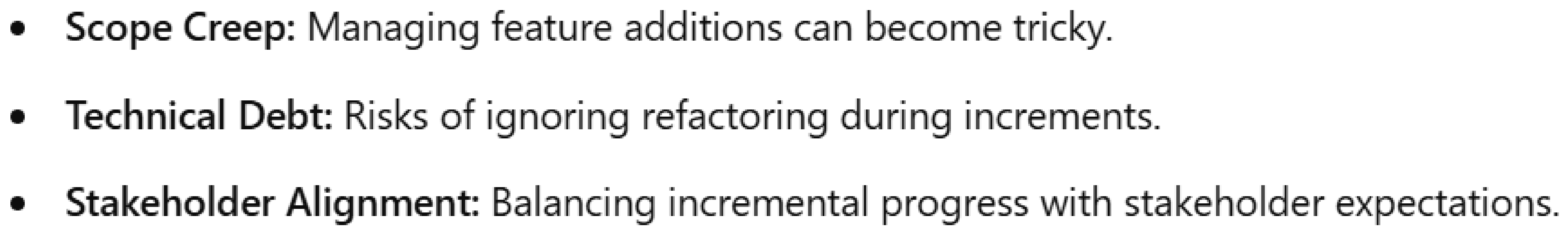


1. **Set Clear Goals:** Define what "valuable increments" mean for your project.
2. **Start with a Thin Slice:** Deliver the smallest useful piece of functionality.
3. **Build on Each Increment:** Continuously refine and expand the product.
4. **Automate Testing:** Ensure every increment meets quality standards.
5. **Engage Stakeholders:** Regularly present increments to gather feedback.

Examples of Incrementalism in Action

- **Agile Development:** Iterative sprints delivering shippable increments.
- **Continuous Delivery Pipelines:** Deploying small changes frequently.
- **Evolving APIs:** Releasing and deprecating API functionalities incrementally.

Challenges in Incremental Development



- **Scope Creep:** Managing feature additions can become tricky.
- **Technical Debt:** Risks of ignoring refactoring during increments.
- **Stakeholder Alignment:** Balancing incremental progress with stakeholder expectations.

Overcoming Challenges

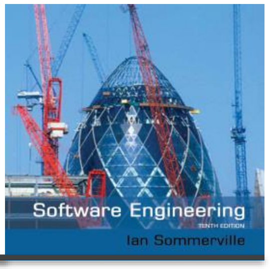
- **Prioritize Ruthlessly:** Keep focus on delivering value.
- **Maintain Quality:** Enforce rigorous testing for every increment.
- **Communication:** Keep stakeholders informed of progress and trade-offs.

Incrementalism and Modern Software Engineering

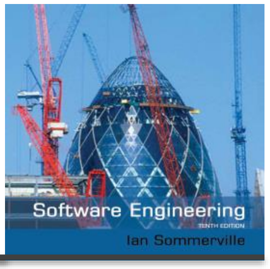
- **Incrementalism is NOT about shortcuts:** It’s about disciplined, continuous progress.
- **Build Iteratively, but Never Compromise Quality:** Deliver small, complete, and valuable features.
- **Aligned with David Farley's Philosophy:**
  - "Doing what works" means adopting practices like CI/CD, TDD, and feature toggles to enable incrementalism.

Conclusion

- Incrementalism emphasizes continuous delivery of value.
- It minimizes risk, accelerates feedback, and improves adaptability.
- **Call to Action:** Start small, iterate fast, and aim for long-term success in your software projects.



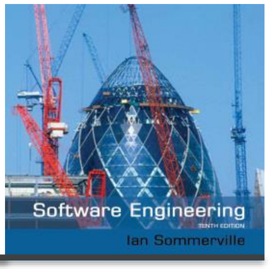
Chapter 7 – Design and Implementation



Design and implementation

- ✧ Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- ✧ Software design and implementation activities are invariably inter-leaved.
  - Software design is a creative activity in which you identify software components and their relationships, based on a customer’s requirements.
  - Implementation is the process of realizing the design as a program.

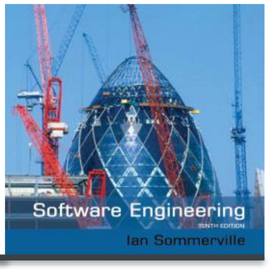
Build or buy



- ✧ In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- ✧ When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

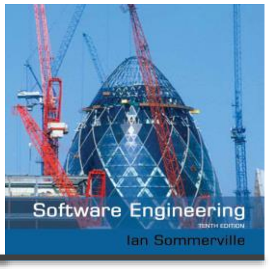
Object-oriented design using the UML

An object-oriented design process



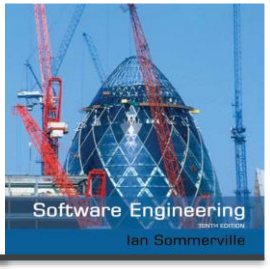
- ✧ Structured object-oriented design processes involve developing a number of different system models.
- ✧ They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- ✧ However, for large systems developed by different groups design models are an important communication mechanism.

Process stages



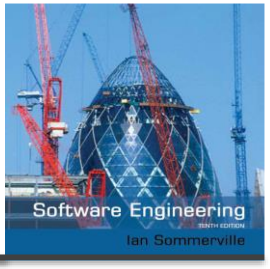
- ✧ There are a variety of different object-oriented design processes that depend on the organization using the process.
- ✧ Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.
- ✧ Process illustrated here using a design for a wilderness weather station.

System context and interactions



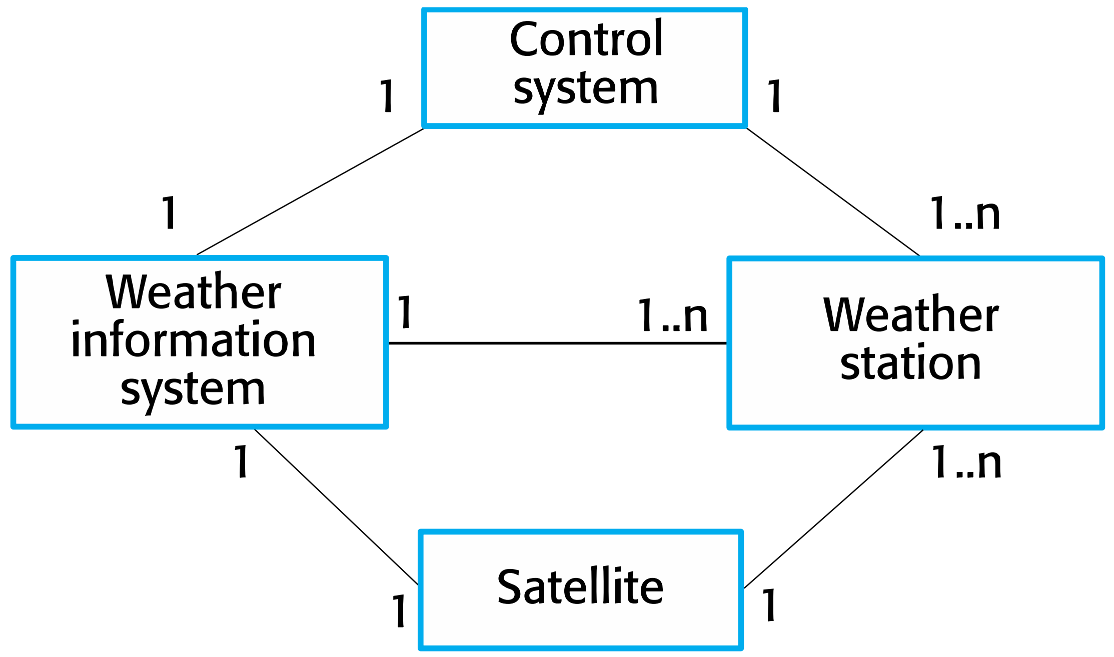
- ✧ Understanding the relationships between the software that is being designed and its external environment is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- ✧ Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

Context and interaction models

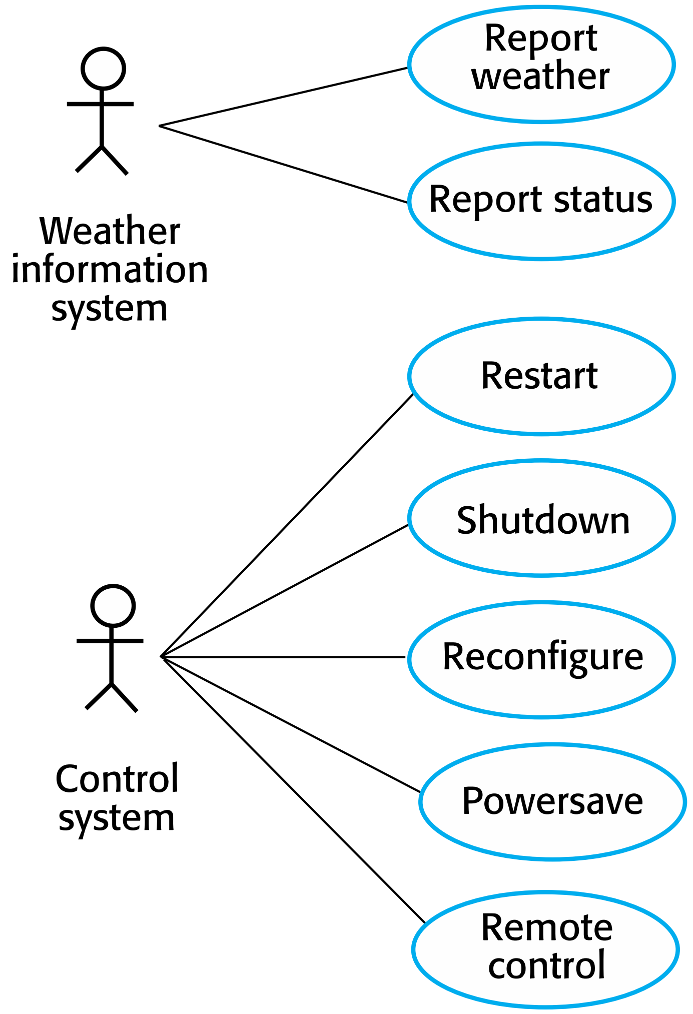


- ✧ A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- ✧ An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

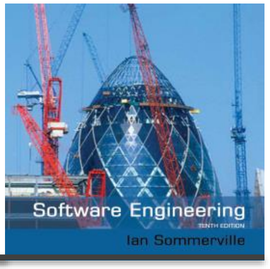
System context for the weather station



Weather station use cases

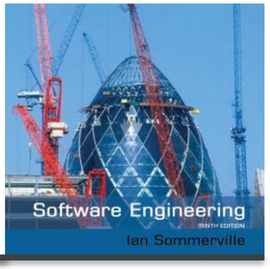


Use case description—Report weather



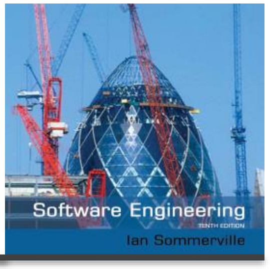
System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, 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 is 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.

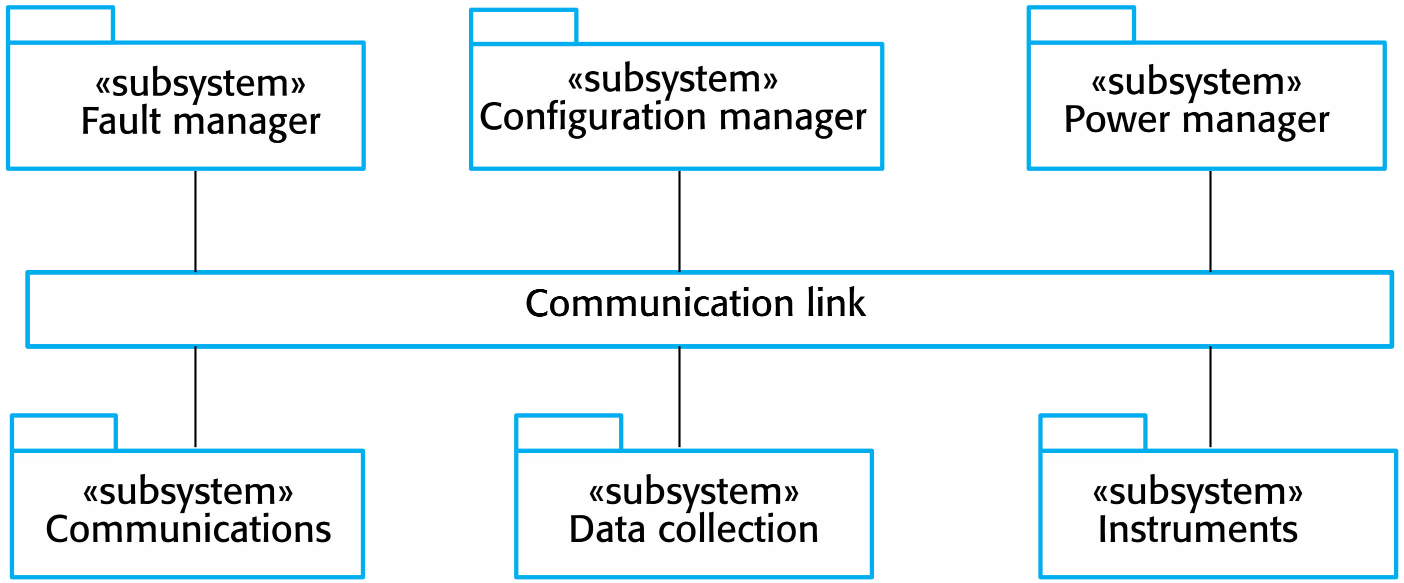
Architectural design



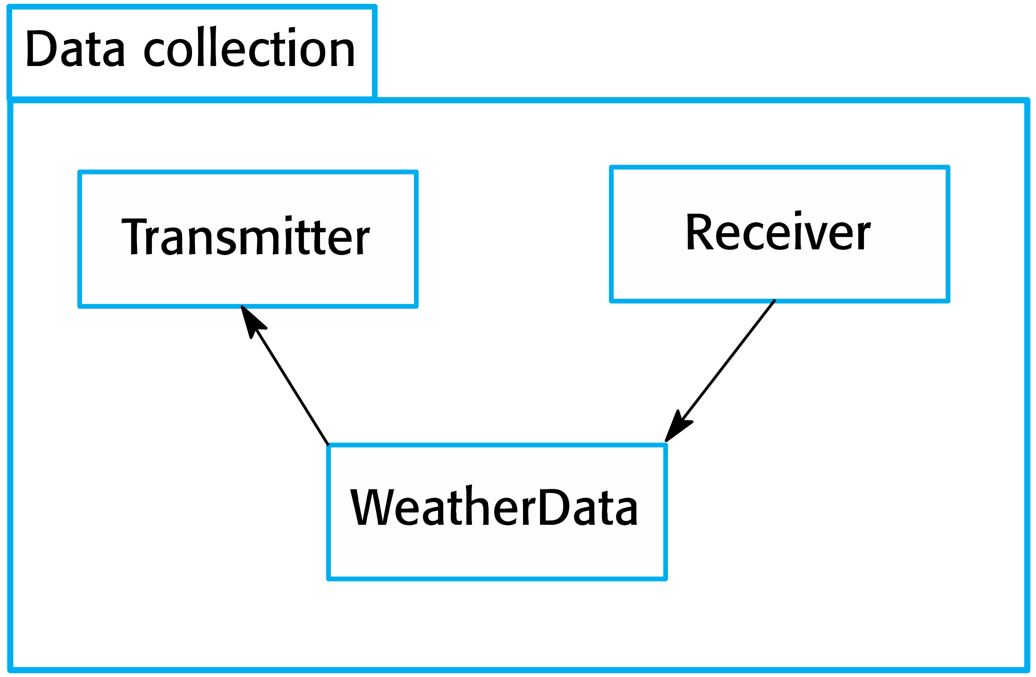
- ✧ Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- ✧ You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
- ✧ The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

High-level architecture of the weather station

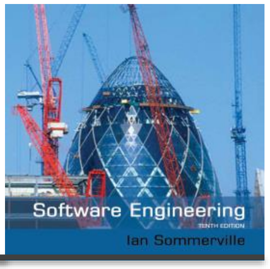




Architecture of data collection system

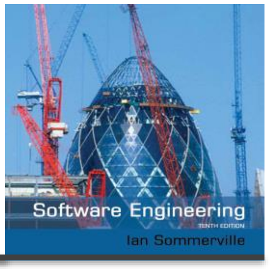


Object class identification



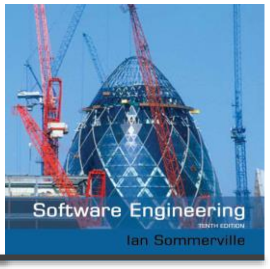
- ✧ Identifying object classes is often a difficult part of object oriented design.
- ✧ There is no 'magic formula' for object identification. It relies on the skill, experience
  and domain knowledge of system designers.
- ✧ Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification



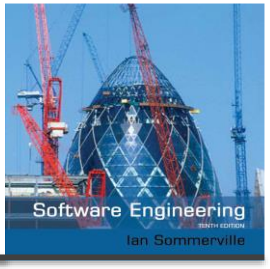
- ✧ Use a grammatical approach based on a natural language description of the system.
- ✧ Base the identification on tangible things in the application domain.
- ✧ Use a behavioural approach and identify objects based on what participates in what behaviour.
- ✧ Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

Design models



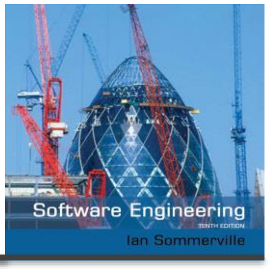
- ✧ Design models show the objects and object classes and relationships between these entities.
- ✧ There are two kinds of design model:
  - Structural models describe the static structure of the system in terms of object classes and relationships.
  - Dynamic models describe the dynamic interactions between objects.

Examples of design models



- ✧ Subsystem models that show logical groupings of objects into coherent subsystems.
- ✧ Sequence models that show the sequence of object interactions.
- ✧ State machine models that show how individual objects change their state in response to events.
- ✧ Other models include use-case models, aggregation models, generalisation models, etc.

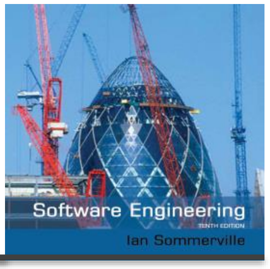
Subsystem models



✧ Shows how the design is organised into logically related groups of objects.

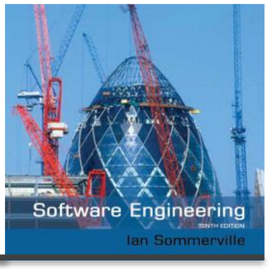
✧ In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

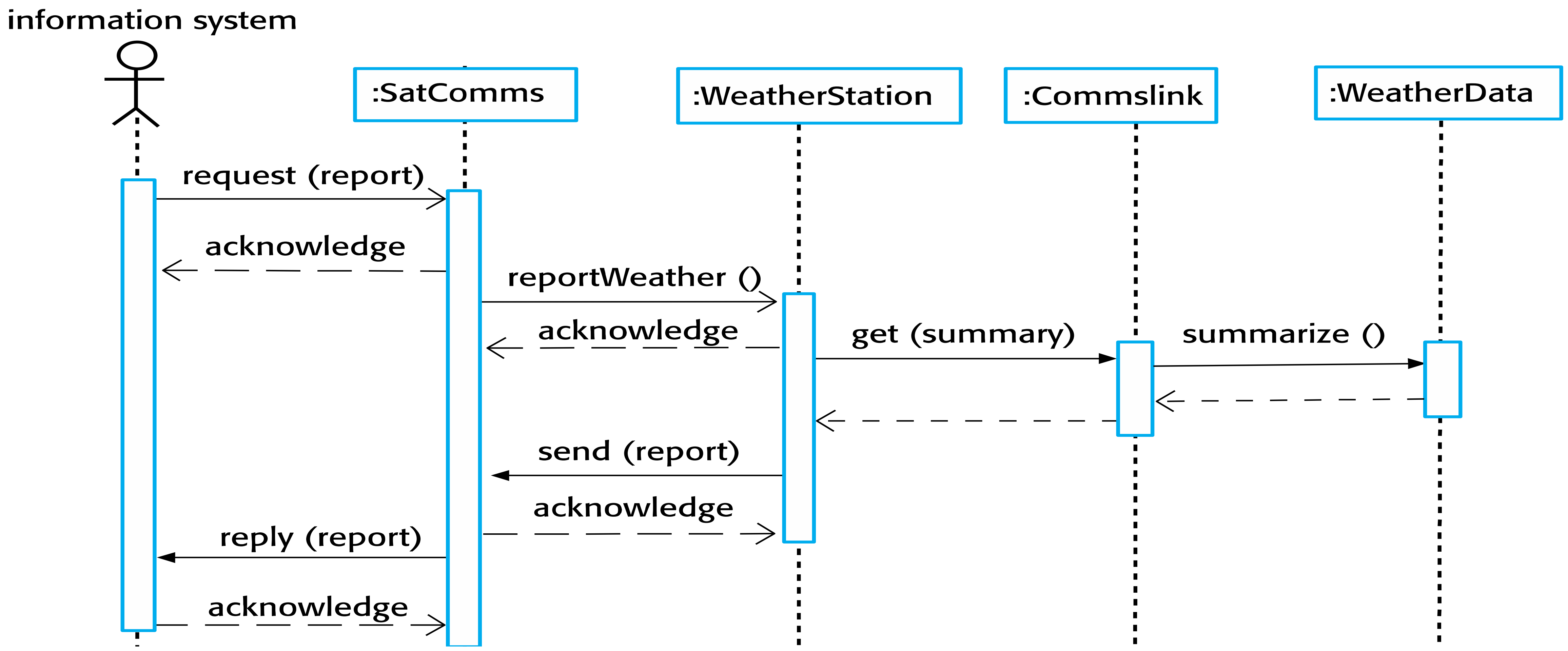
Sequence models



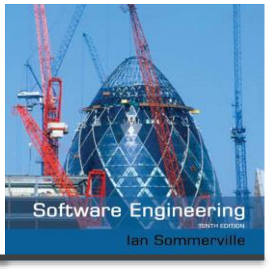
- ✧ Sequence models show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

Sequence diagram describing data collection



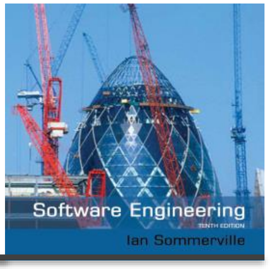


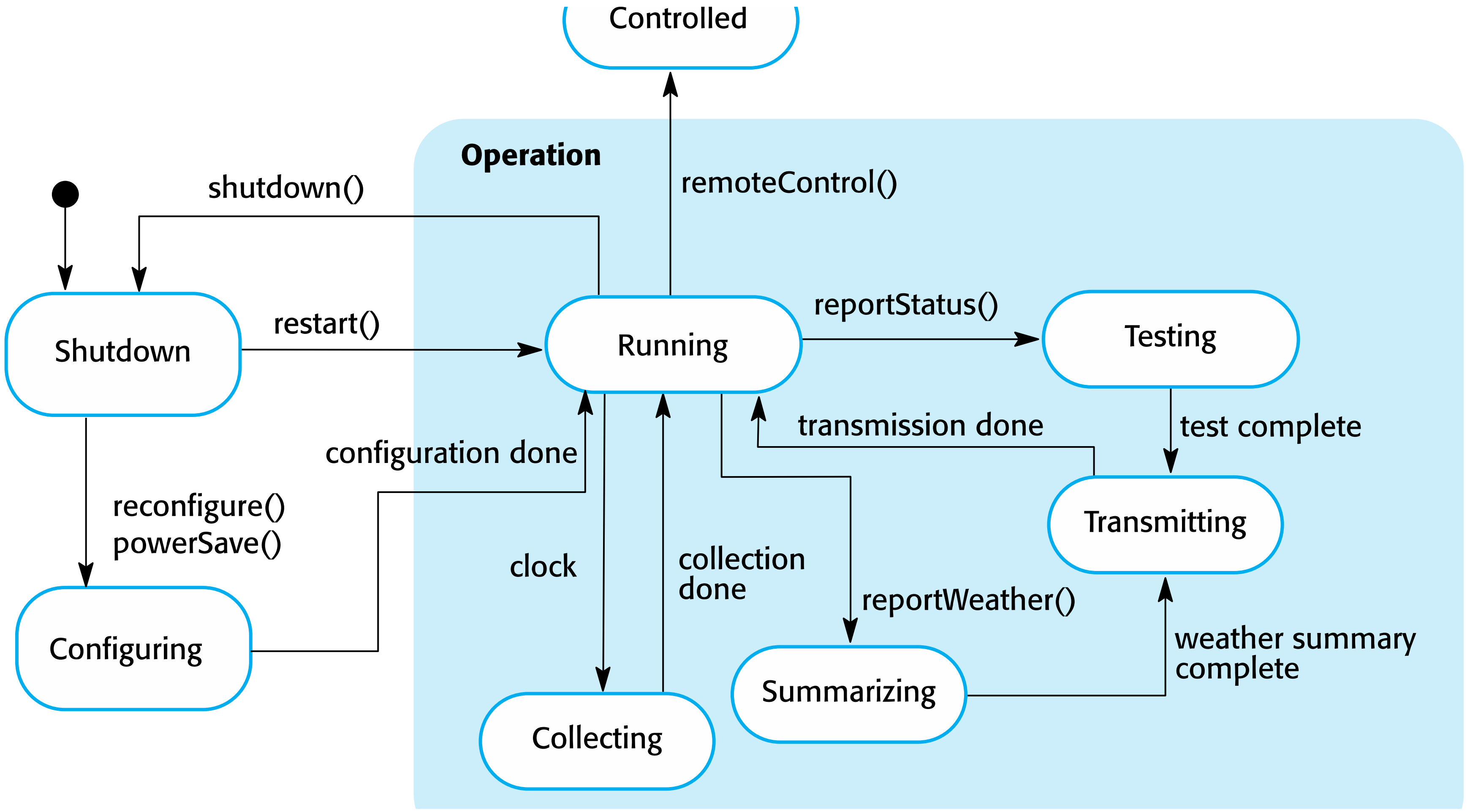
State diagrams



- ✧ State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- ✧ State diagrams are useful high-level models of a system or an object’s run-time behavior.
- ✧ You don’t usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

Weather station state diagram

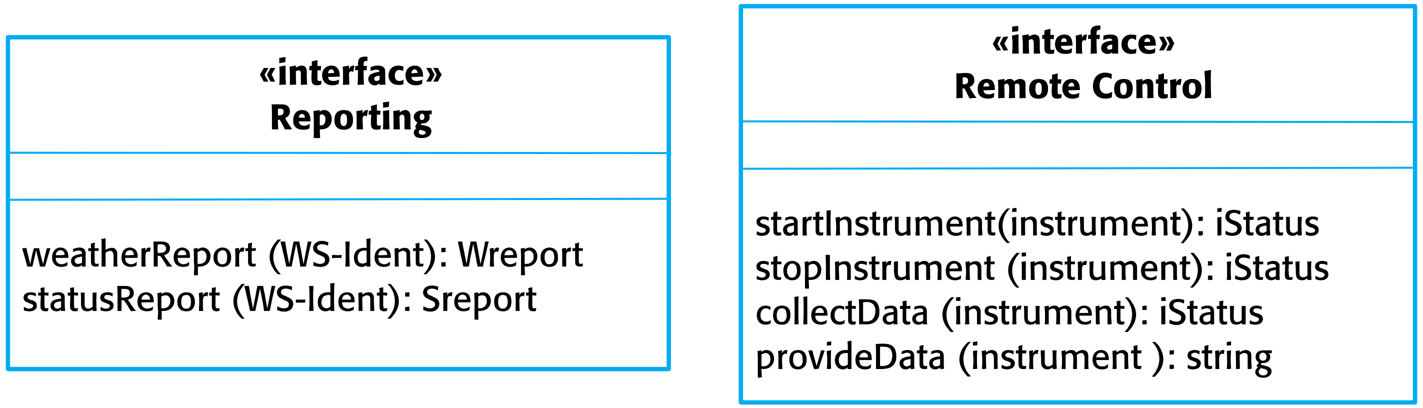




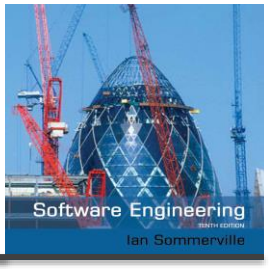
Interface specification

- ✧ Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- ✧ Designers should avoid designing the interface representation but should hide this in the object itself.
- ✧ Objects may have several interfaces which are viewpoints on the methods provided.
- ✧ The UML uses class diagrams for interface specification but Java may also be used.

Weather station interfaces


«interface»
Reporting
weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport
«interface»
Remote Control
startInstrument(instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument): string

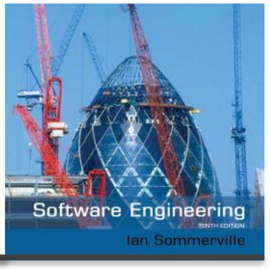
Design patterns



Design patterns

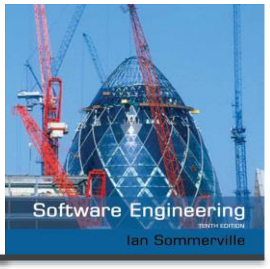
- ✧ A design pattern is a way of reusing abstract knowledge about a problem and its solution.
- ✧ A pattern is a description of the problem and the essence of its solution.
- ✧ It should be sufficiently abstract to be reused in different settings.
- ✧ Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

Patterns



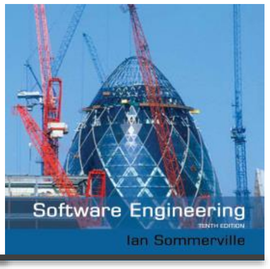
✧ *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience.*

Pattern elements



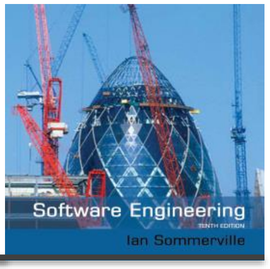
- ✧ Name
  - A meaningful pattern identifier.
- ✧ Problem description.
- ✧ Solution description.
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- ✧ Consequences
  - The results and trade-offs of applying the pattern.

The Observer pattern



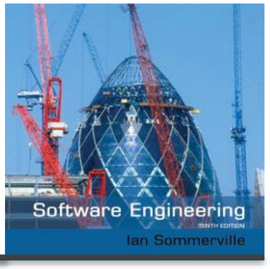
- ✧ Name
  - Observer.
- ✧ Description
  - Separates the display of object state from the object itself.
- ✧ Problem description
  - Used when multiple displays of state are needed.
- ✧ Solution description
  - See slide with UML description.
- ✧ Consequences
  - Optimisations to enhance display performance are impractical.

The Observer pattern (1)



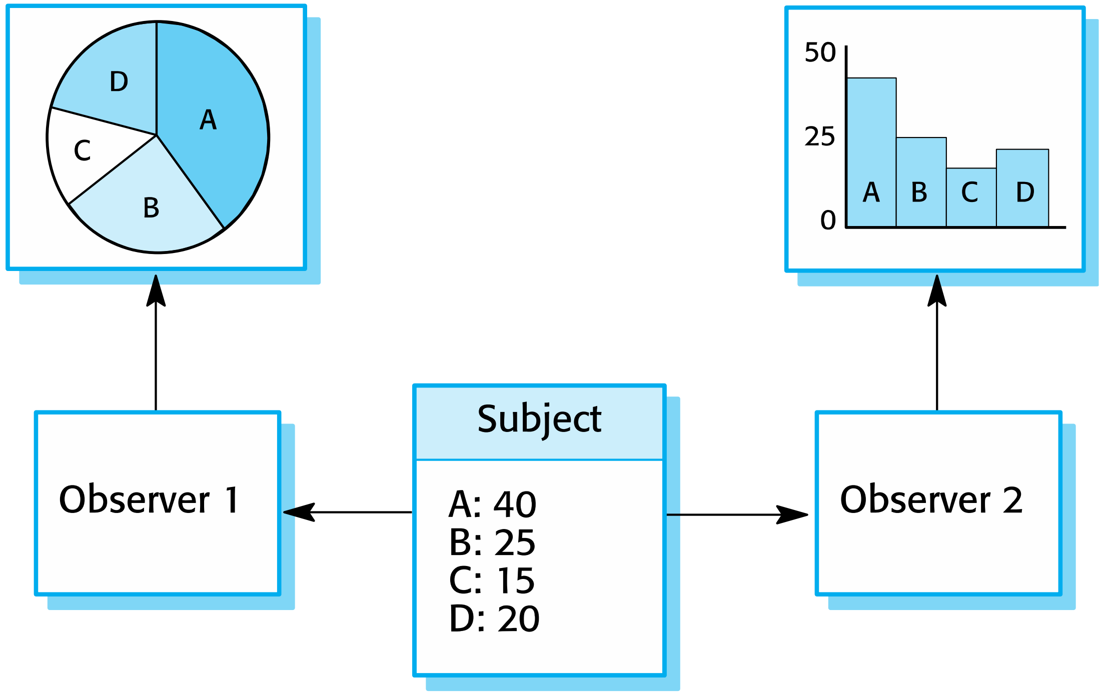
Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

The Observer pattern (2)

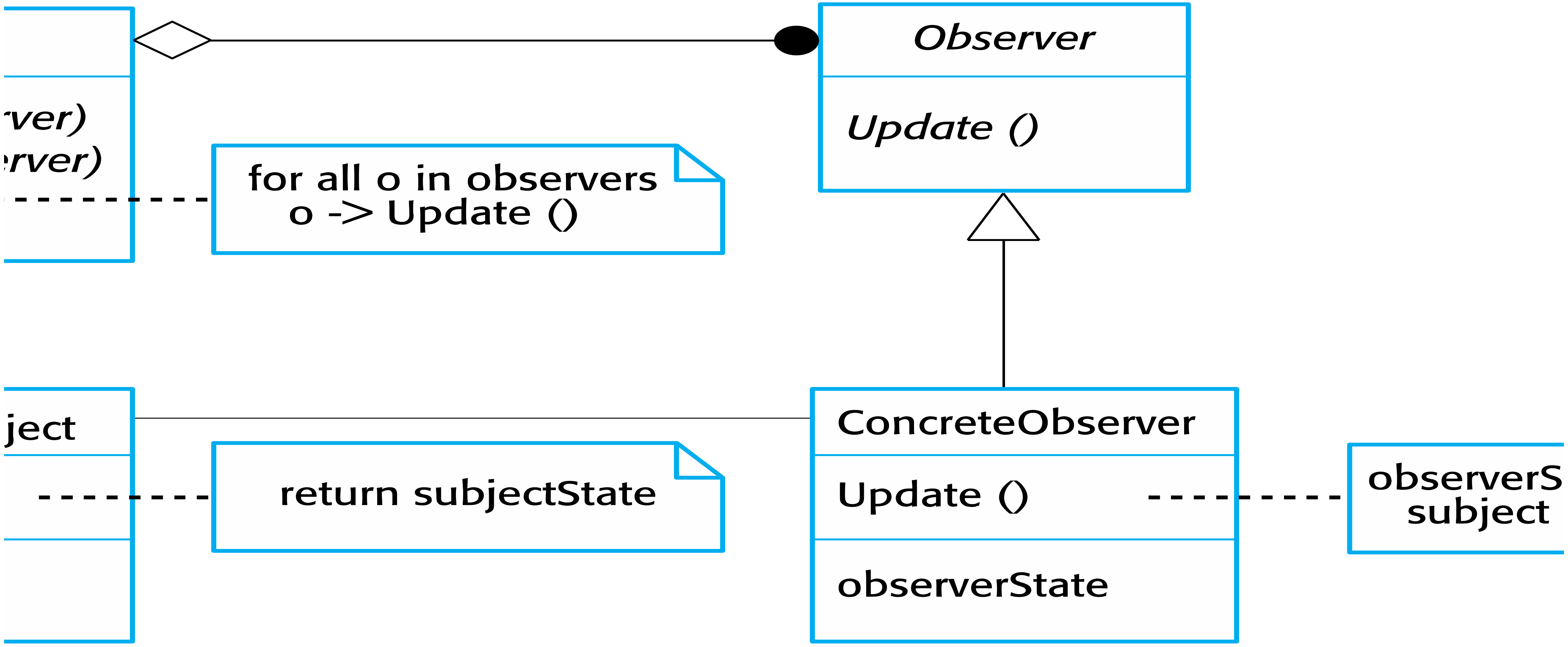


Pattern name	Observer
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Multiple displays using the Observer pattern



A UML model of the Observer pattern



Achieving Quality Attributes

Main Quality Attributes

1. Modifiability
2. Performance
3. Security
4. Reliability
5. Robustness
6. Usability
7. Business Goals

Modifiability

- More than 50% of cost occurs after the first release!
- Change: Direct changes + Indirect changes
- 100 modules – only a few will undergo direct change
- How to code in such a way that changes are minimized?
- Minimize Direct Changes
  - Anticipate the change + Ensure High Cohesion + Generalize
- Minimize Indirect Changes
  - Ensure Low Coupling + Stable Interface + Have Multiple Interfaces

Performance

- Response Time (how fast) + Throughput (how many requests per second) + Load (how many users)
- Tips
  - Have more resources (faster processor + more memory + more bandwidth)
  - Code to better utilize resources (concurrency)
  - Better manage resource allocation – FCFS, prioritize
  - Reduce demand for resources

Security

- Immunity (will you be attacked?) and Resilience (how quickly you can recover)
- Ensure High Immunity
  - Include all security features in design
  - Minimize security weakness
- Ensure high Resilience
  - Segment the code
  - Restore data and functionality

Reliability and Robustness

- Correctly performs under the adverse conditions
- Reliability versus Robustness
  - Reliability = works well under expected conditions
  - Robustness = works well under unexpected conditions
- Reliability
  - Detect fault
  - Recover from fault
    - Undo, Rollback, Backup, Offer degraded service, Correct and Continue, Report
- Robustness
  - Policy of mutual suspicion

Usability

- Ease of use of your software
- Tips
  - Good UI design
  - Good UX design
  - Separate module for UI
  - Provide cancel, undo, ….
  - Pilot (ask users for feedback)
  - Regular Feedback

Business Goals

- Buy versus Build
- Minimize costs of maintenance
- Minimize surprises to end users
- New versus known technologies

Implementation

Coding

Implementation - Steps

1. Standards of Programming
2. Guidelines for Reuse
3. Organize the code based on the design
4. Documentation: Internal and External

(1) Programming/Coding Standard

- Indentation (Spacing)
- Inline comments
- Use of globals
- Structured programming
- Naming conventions
- Errors and Exception Handling

- Goals of Standards
  - Make the code readable
  - Make the code robust
  - Make the code consistent

(2) Reuse

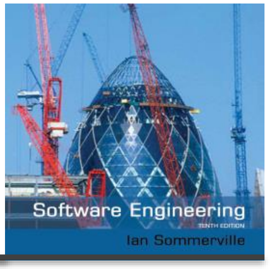
1. Systems can be reused (Ex. Ticket Booking System)
2. Reuse components (Ex. Shopping Cart)
3. Reuse software libraries (Ex. Python libraries like Numpy, Scipy)
4. Reuse abstract concept (Ex. MVC pattern)

(3) Organize Code Based on Design

1. Follow the design to organize the code
2. Entire project = Root folder
3. Packages = Separate sub-directory under Root
4. Component = Sub Sub Directory
5. Class = Separate File
6. Utilities (like JS files), Images go into their own folders

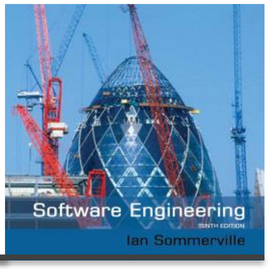
(4) Documentation

- Highly neglected
- Types:
  - Internal documentation
    - comments inside the code
    - Blocks at the top of the code
    - Comments before a code block or condition
    - Inline comments
  - External Documentation
    - Maintained outside the code



Chapter 8 – Software Testing

Topics covered



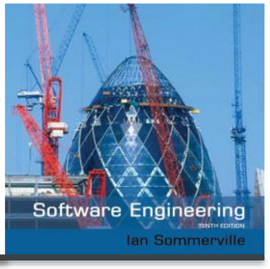
✧ Development testing

✧ 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.

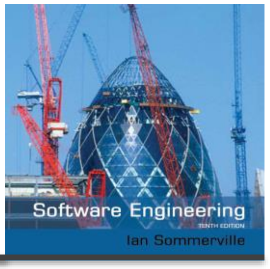
✧ When you test software, you execute a program using artificial data.

✧ 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.

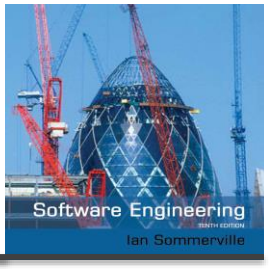
✧ Testing is part of a more general verification and validation process, which also includes static validation techniques.

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



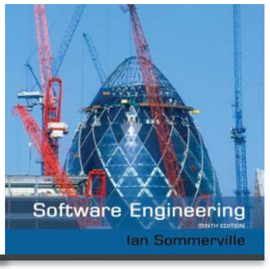
✧ The first goal leads to validation testing

- 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



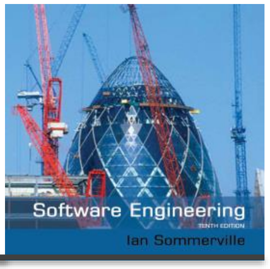
✧ Validation testing

- 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

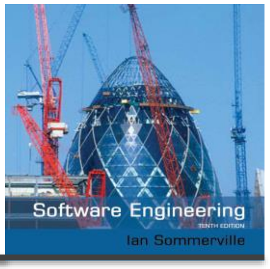
- To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification
- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Verification vs validation



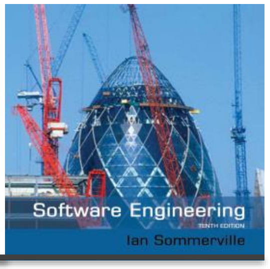
- ✧ Verification:
  "Are we building the product right”.
- ✧ The software should conform to its specification.
- ✧ Validation:
  "Are we building the right product”.
- ✧ The software should do what the user really requires.

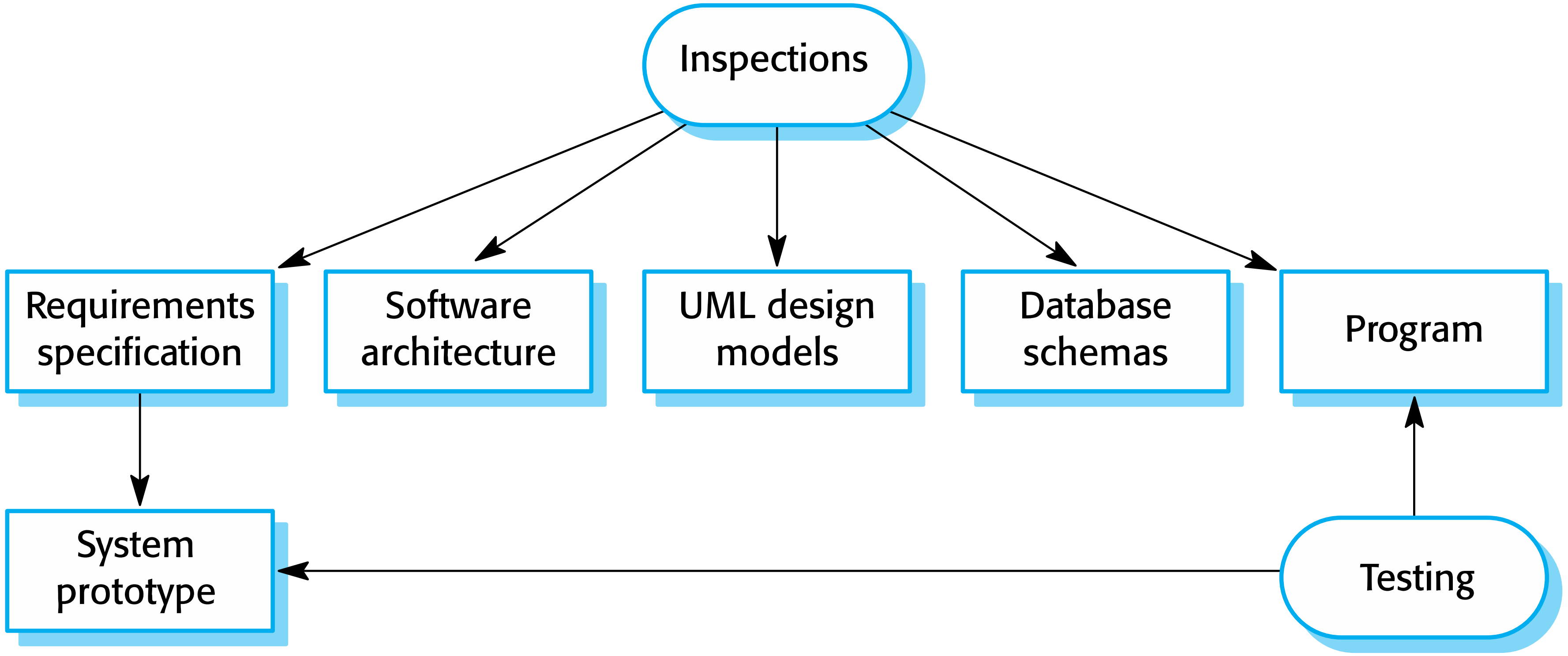
Inspections and testing



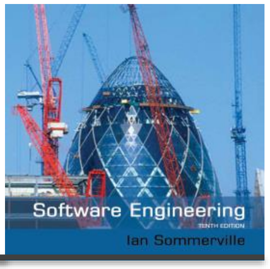
- ✧ Software inspections Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis.
  - Discussed in Chapter 15.
- ✧ Software testing Concerned with exercising and observing product behaviour (dynamic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

Inspections and testing



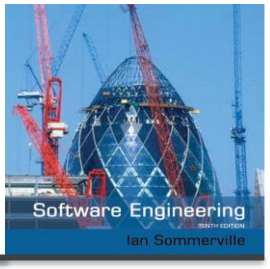


Software inspections



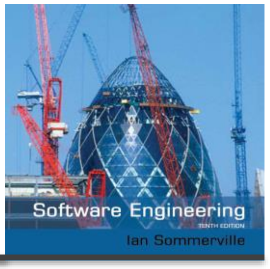
- ✧ These involve people examining the source representation with the aim of discovering anomalies and defects.
- ✧ Inspections not require execution of a system so may be used before implementation.
- ✧ They may be applied to any representation of the system (requirements, design,configuration data,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

Advantages of inspections



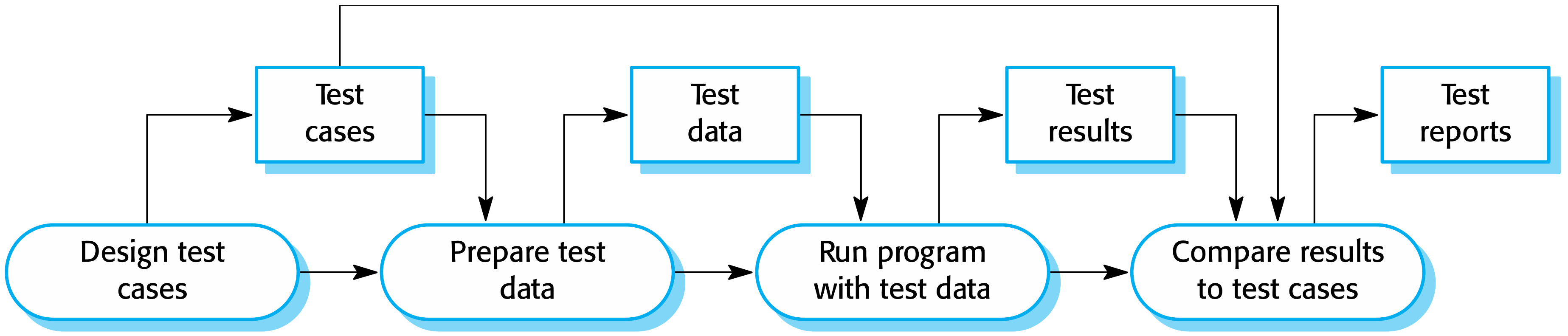
- ✧ During testing, errors can mask (hide) other errors. Because inspection is a static process, you don’t have to be concerned with interactions between errors.
- ✧ Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- ✧ As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

Inspections and testing

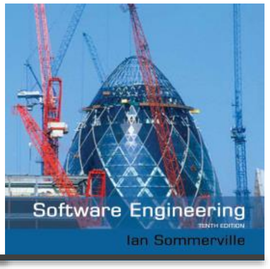


- ✧ Inspections and testing are complementary and not opposing verification techniques.
- ✧ Both should be used during the V & V process.
- ✧ Inspections can check conformance with a specification but not conformance with the customer’s real requirements.
- ✧ Inspections cannot check non-functional characteristics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

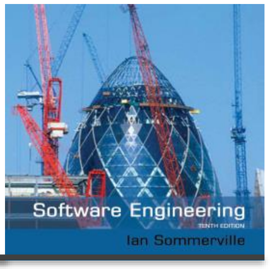
A model of the software testing process



Stages of testing

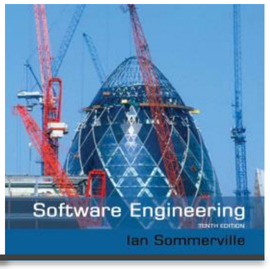


- ✧ Development testing, where the system is tested during development to discover bugs and defects.
- ✧ Release testing, where a separate testing team test a complete version of the system before it is released to users.
- ✧ User testing, where users or potential users of a system test the system in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



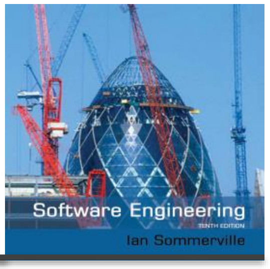
Development testing

Development testing



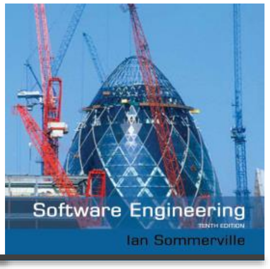
- Development testing includes all testing activities that are carried out by the team developing the system.
  - Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  - Component testing, where several 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



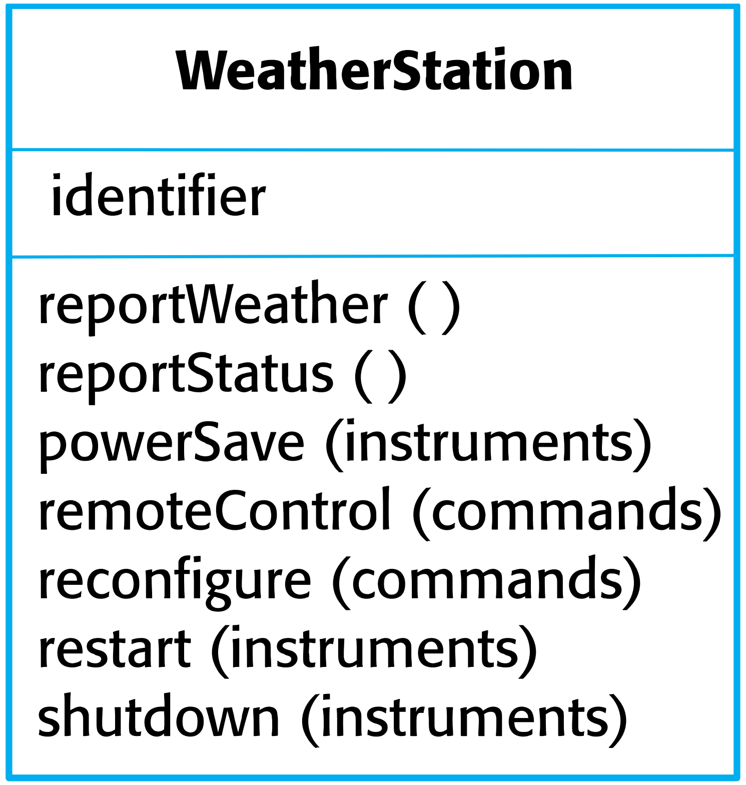
- ✧ Unit testing is the process of testing individual components in isolation.
- ✧ It is a defect testing process.
- ✧ Units may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components with defined interfaces used to access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

Object class testing

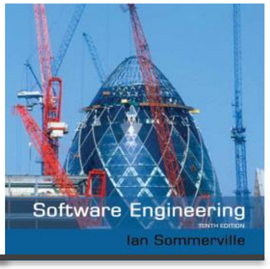


- ✧ Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states.
- ✧ Inheritance makes it more difficult to design object class tests as the information to be tested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

The weather station object interface

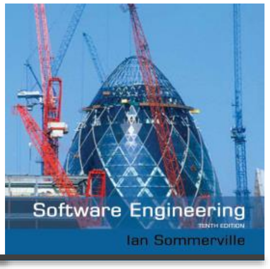


Weather station testing



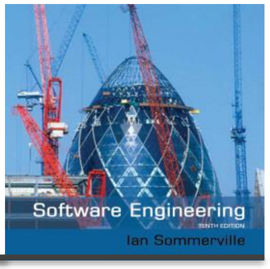
- ✧ Need to define test cases for reportWeather, calibrate, test, startup and shutdown.
- ✧ Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions
- ✧ For example:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing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

Testing strategies



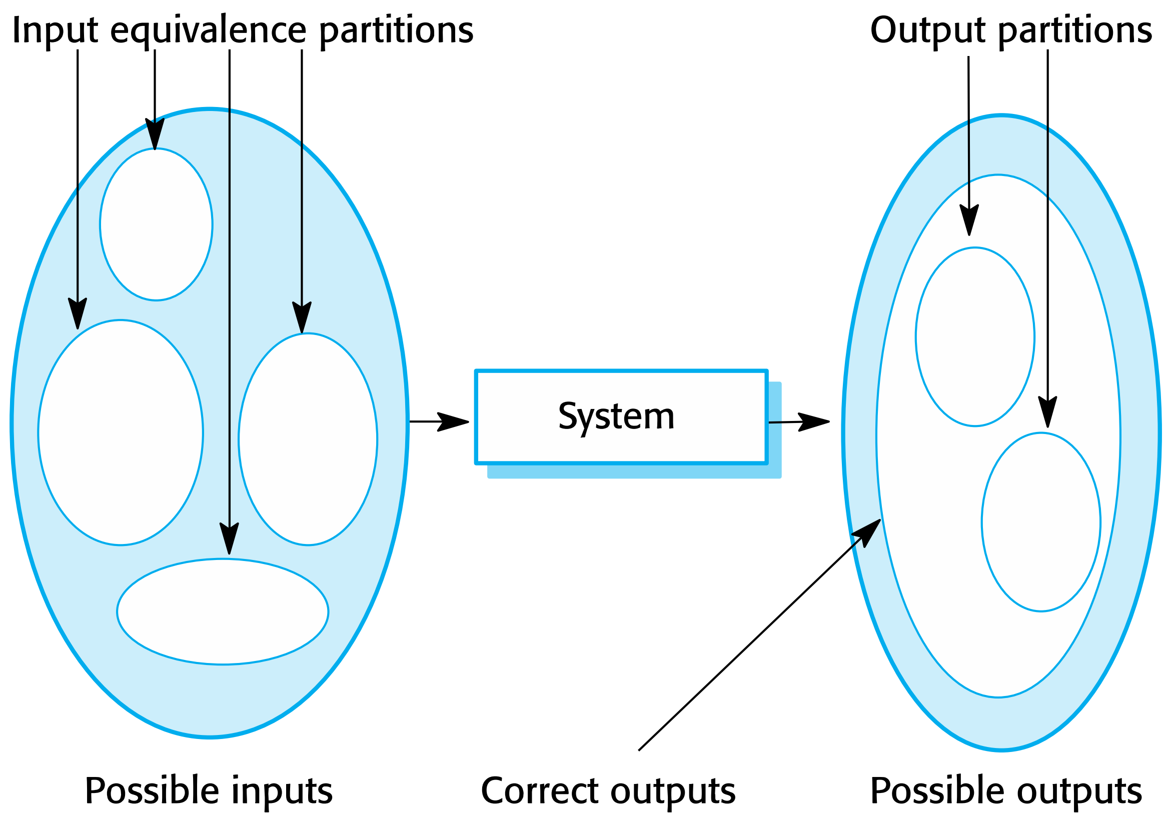
- ✧ Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
  - You should choose tests from within each of these groups.
- ✧ Guideline-based testing, where you use testing guidelines to choose test cases.
  - These guidelines reflect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

Partition testing

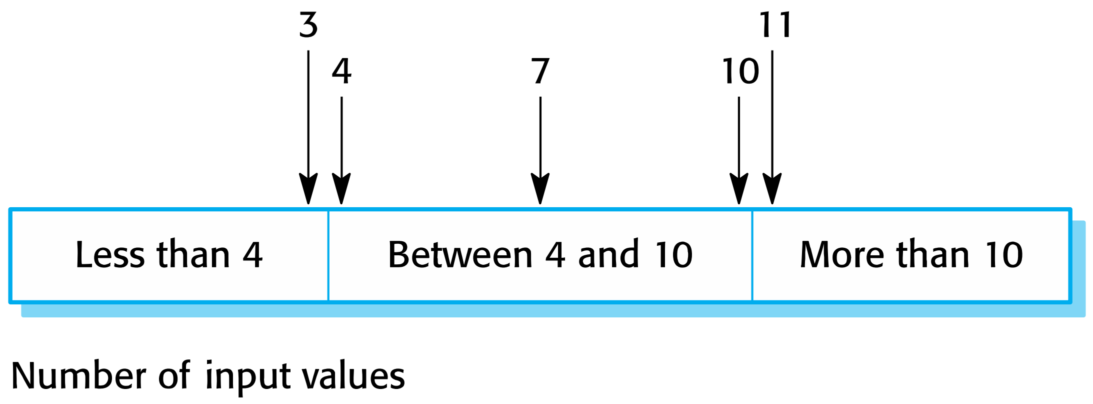


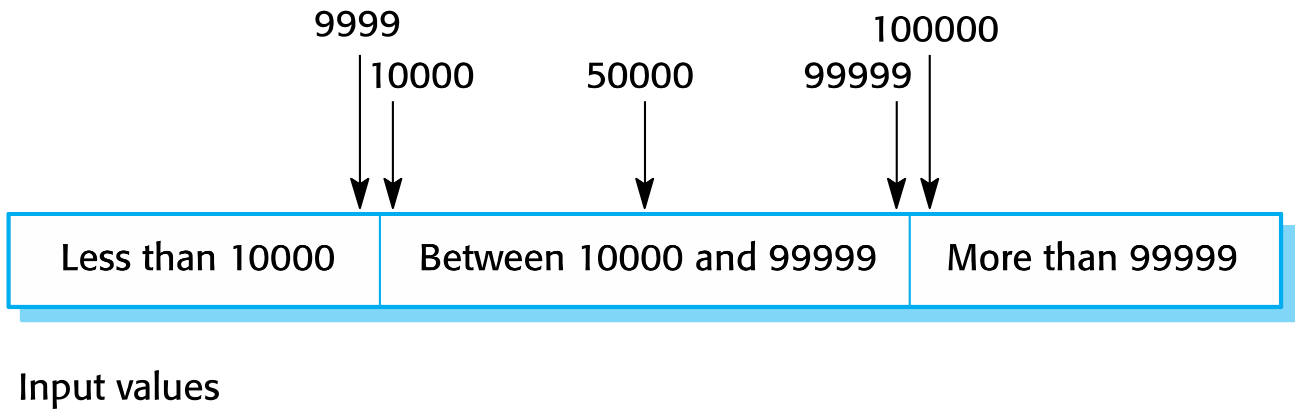
- ✧ Input data and output results often fall into different classes where all members of a class are related.
- ✧ Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- ✧ Test cases should be chosen from each partition.

Equivalence partitioning

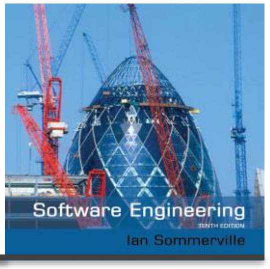


Equivalence partitions



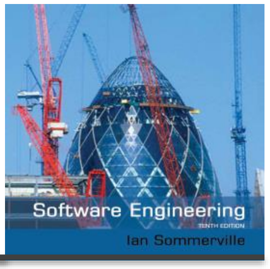


Testing guidelines (sequences)



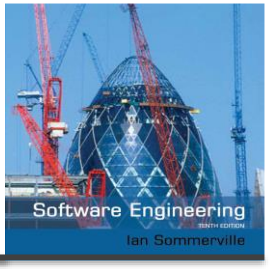
- ✧ Test software with sequences which have only a single value.
- ✧ Use sequences of different sizes in different tests.
- ✧ Derive tests so that the first, middle and last elements of the sequence are accessed.
- ✧ Test with sequences of zero length.

General testing guidelines



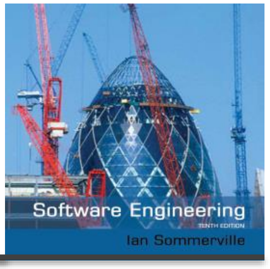
- ✧ Choose inputs that force the system to generate all error messages
- ✧ Design inputs that cause input buffers to overflow
- ✧ Repeat the same input or series of inputs numerous times
- ✧ Force invalid outputs to be generated
- ✧ Force computation results to be too large or too small.

Component testing



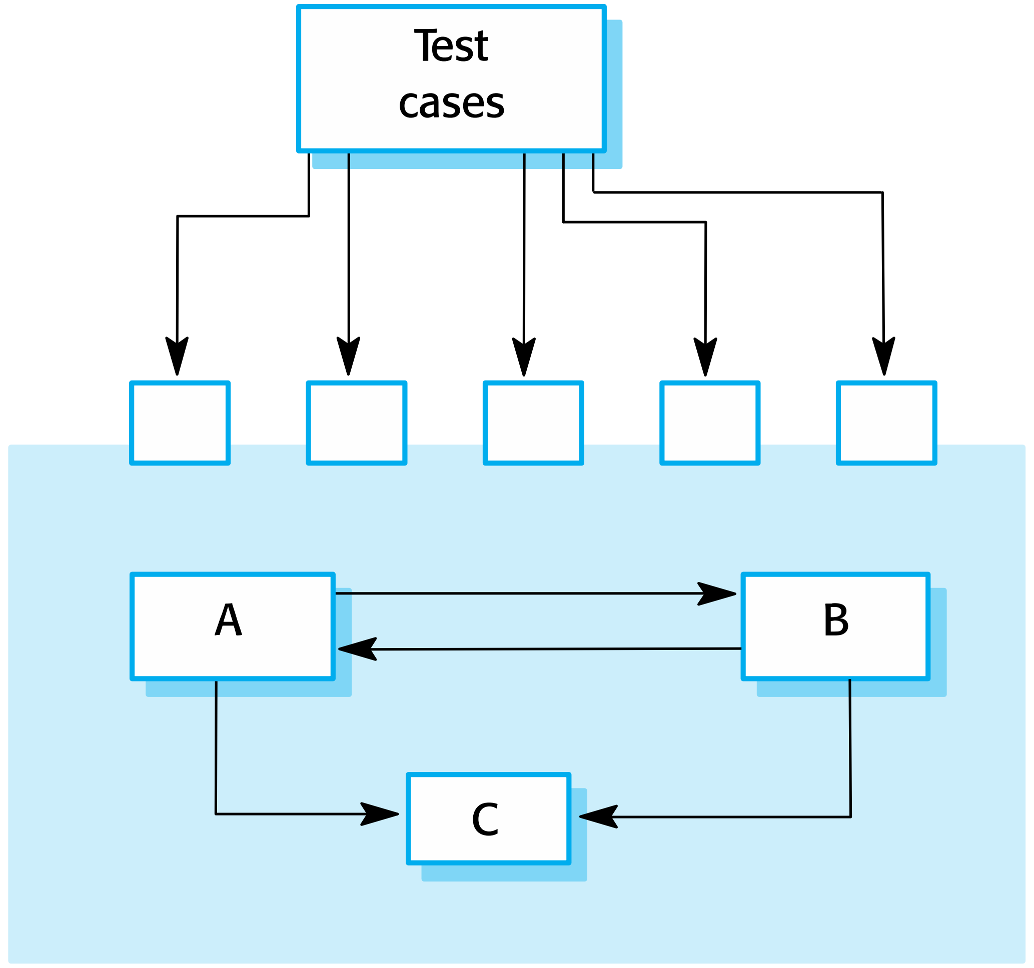
- ✧ Software components are often composite components that are made up of several interacting objects.
  - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- ✧ You access the functionality of these objects through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

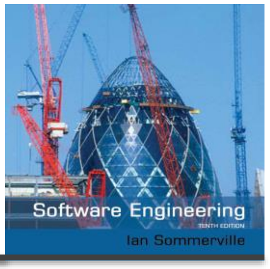


- ✧ Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- ✧ Interface types
  - Parameter interfaces Data passed from one method or procedure to another.
  - Shared memory interfaces Block of memory is shared between procedures or functions.
  - Procedural interfaces Sub-system encapsulates a set of procedures to be called by other sub-systems.
  - Message passing interfaces Sub-systems request services from other sub-systems

Interface testing



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.

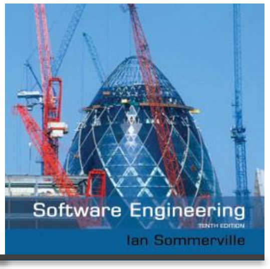
✧ Interface misunderstanding

- A calling component embeds assumptions about the behaviour of the called component which are incorrect.

✧ Timing errors

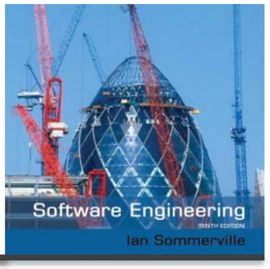
- The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface testing guidelines



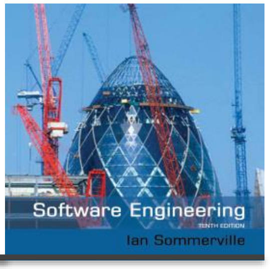
- ✧ Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- ✧ Always test pointer parameters with null pointers.
- ✧ Design tests which cause the component to fail.
- ✧ Use stress testing in message passing systems.
- ✧ In shared memory systems, vary the order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

System testing



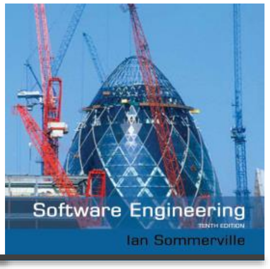
- ✧ System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- ✧ The focus in system testing is testing the interactions between components.
- ✧ System testing checks that components are compatible, interact correctly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

System and component testing



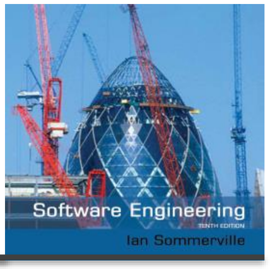
- ✧ During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- ✧ Components developed by different team members or sub-teams may be integrated at this stage. System testing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

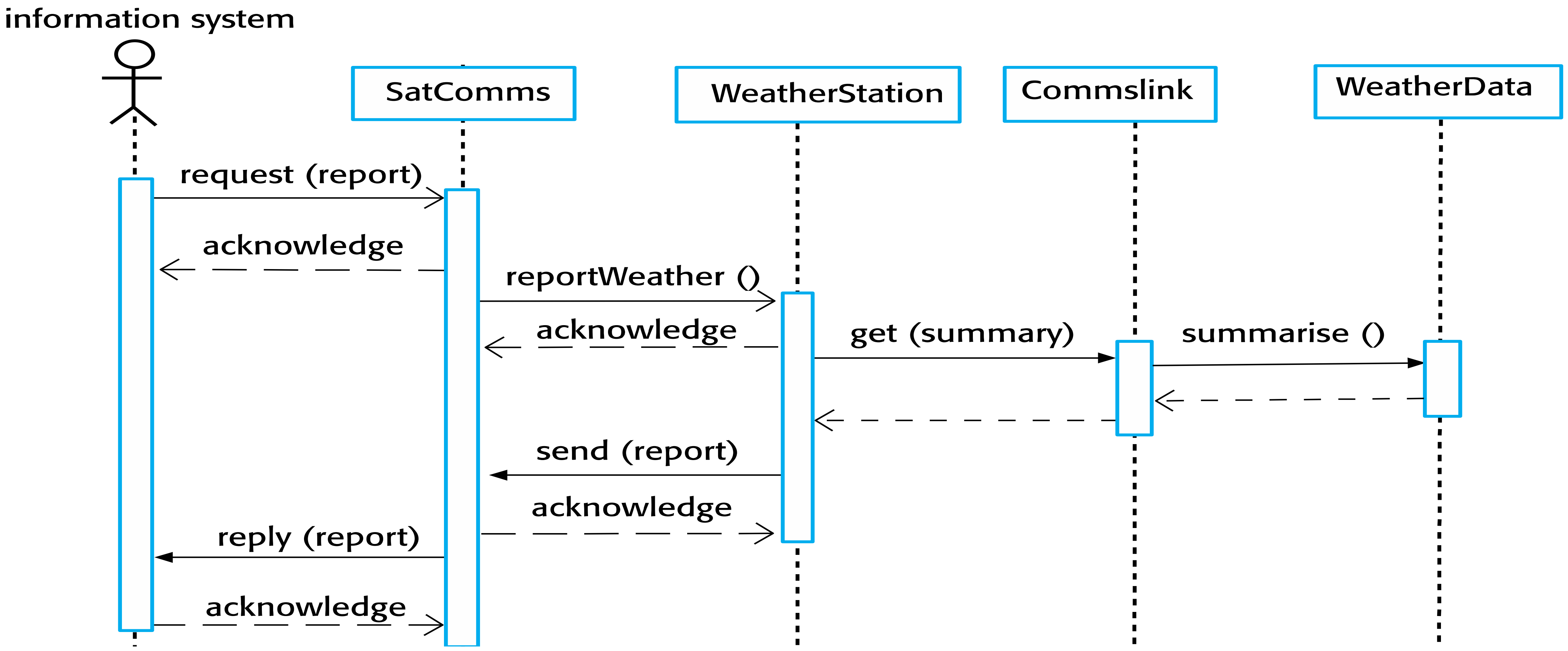
Use-case testing



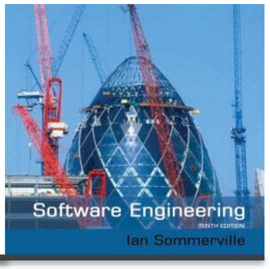
- ✧ The use-cases developed to identify system interactions can be used as a basis for system testing.
- ✧ Each use case usually involves several system components so testing the use case forces these interactions to occur.
- ✧ The sequence diagrams associated with the use case documents the components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

Collect weather data sequence chart



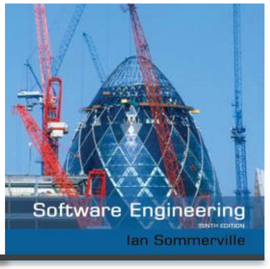


Test cases derived from sequence diagram

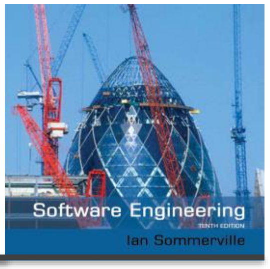


- ✧ An input of a request for a report should have an associated acknowledgement. A report should ultimately be returned from the request.
  - You should create summarized data that can be used to check that the report is correctly organized.
- ✧ An input request for a report to WeatherStation results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

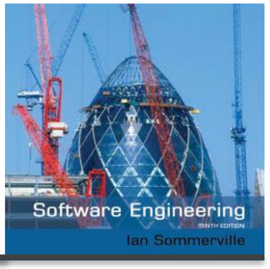


- ✧ Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- ✧ Examples of testing policies:
  - All system functions that are accessed through menus should be tested.
  - Combinations of functions (e.g. text formatting) that are accessed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



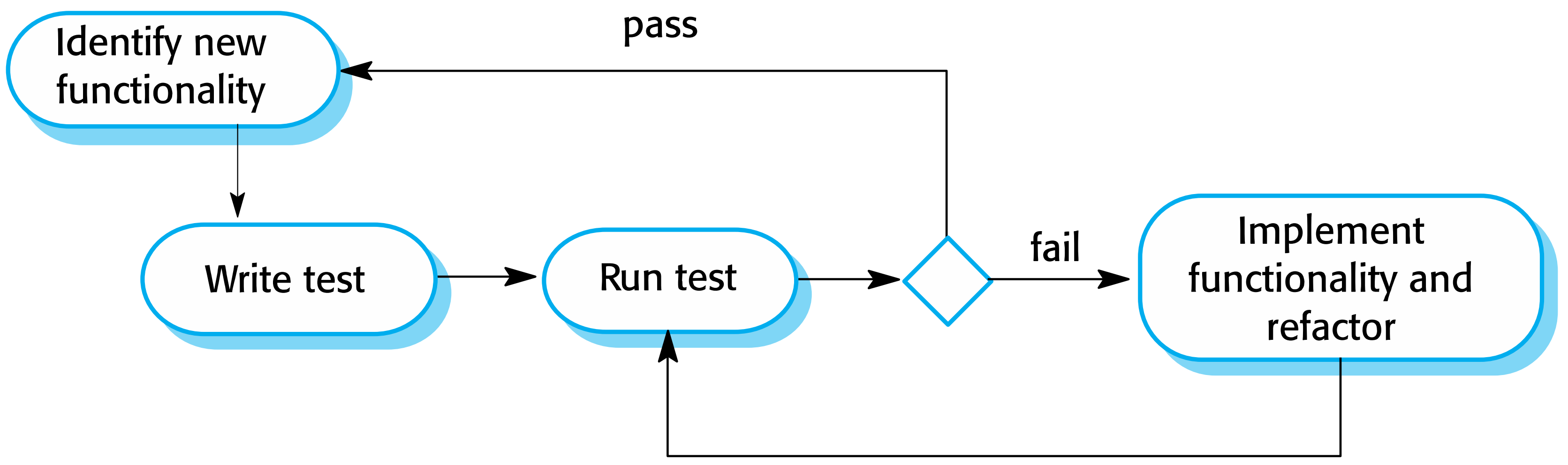
Test-driven development

Test-driven development



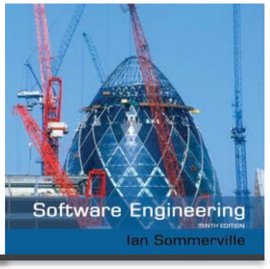
- ✧ Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- ✧ Tests are written before code and 'passing' the tests is the critical driver of development.
- ✧ You develop code incrementally, along with a test for that increment. You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

Test-driven development



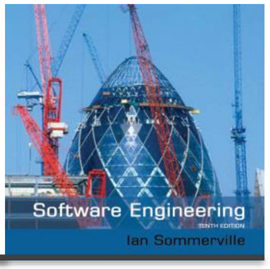
pass
Identify new functionality
Write test
Run test
fail
Implement functionality and refactor

TDD process activities



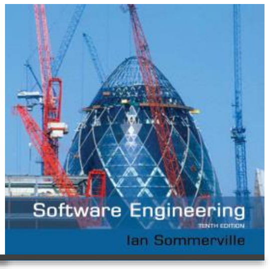
- ✧ Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- ✧ Write a test for this functionality and implement this as an automated test.
- ✧ Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- ✧ Implement the functionality and re-run the test.
- ✧ Once all tests run successfully, you move on to implementing the next chunk of functionality.

Benefits of test-driven development

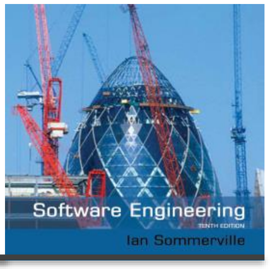


- ✧ Code coverage
  - Every code segment that you write has at least one associated test so all code written has at least one test.
- ✧ Regression testing
  - A regression test suite is developed incrementally as a program is developed.
- ✧ Simplified debugging
  - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- ✧ System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing.

Regression testing

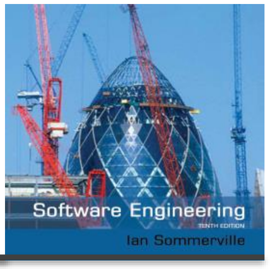


- ✧ Regression testing is testing the system to check that changes have not 'broken' previously working code.
- ✧ In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program.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



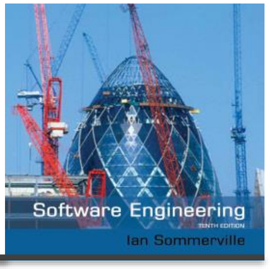
Release testing

Release testing



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

Release testing and system testing

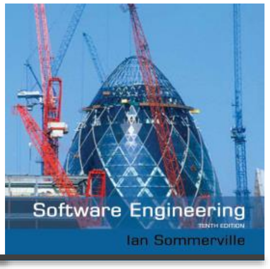


✧ Release testing is a form of system testing.

✧ Important differences:

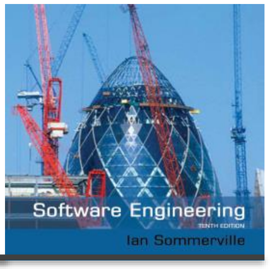
- A separate team that has not been involved in the system development, should be responsible for release testing.
- System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements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

Requirements based testing



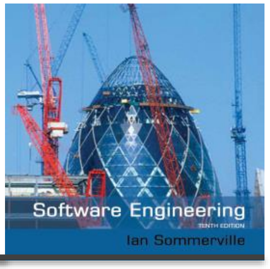
- ✧ Requirements-based testing involves examining each requirement and developing a test or tests for it.
- ✧ Mentcare system requirements:
  - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the 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.

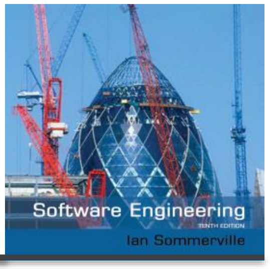
Performance testing



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

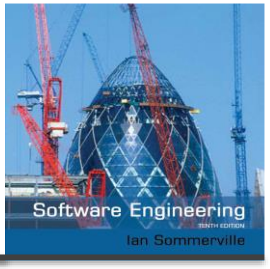
User testing

User testing



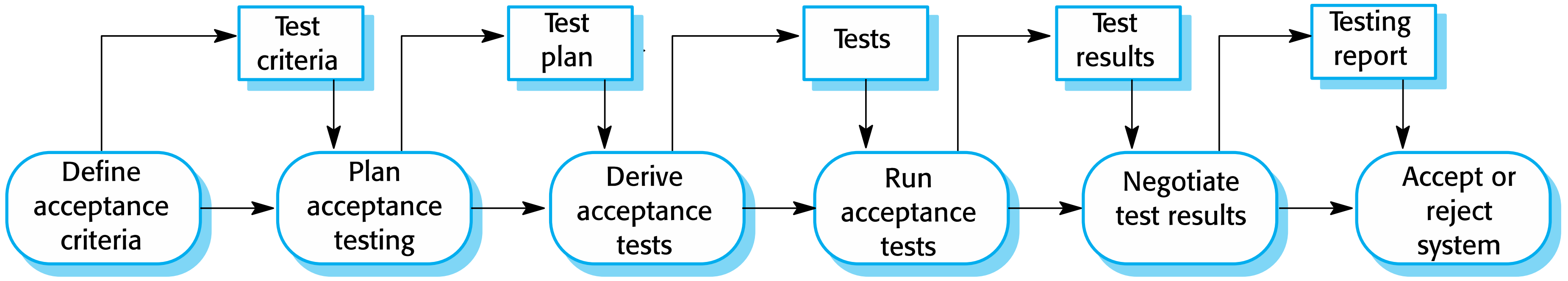
- ✧ User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.
- ✧ User testing is essential, even when comprehensive system and release testing have been carried out.
  - ▪ The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

Types of user testing

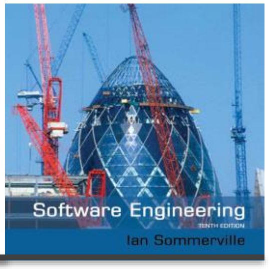


- ✧ Alpha testing
  - Users of the software work with the development team to test the software at the developer's site.
- ✧ Beta testing
  - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
- ✧ Acceptance testing
  - Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

The acceptance testing process

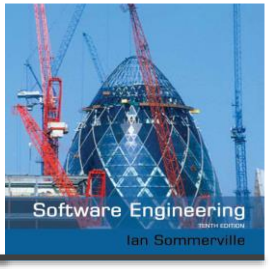


Stages in the acceptance testing process



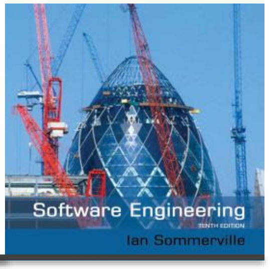
- ✧ Define acceptance criteria
- ✧ Plan acceptance testing
- ✧ Derive acceptance tests
- ✧ Run acceptance tests
- ✧ Negotiate test results
- ✧ Reject/accept system

Agile methods and acceptance testing



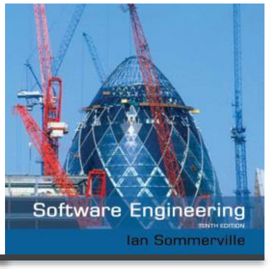
- ✧ In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system.
- ✧ Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- ✧ There is no 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

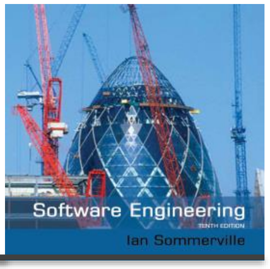


- ✧ Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- ✧ Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.
- ✧ Development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

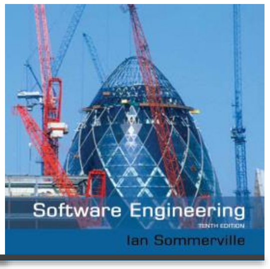


- ✧ When testing software, you should try to ‘break’ the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.
- ✧ Wherever possible, you should write automated tests. The tests are embedded in a program that can be 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.



Chapter 9 – Software Evolution

Topics covered

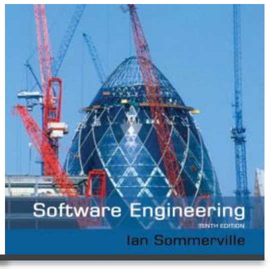


✧ Evolution processes

✧ Legacy systems

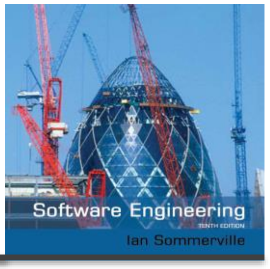
✧ Software maintenance

Software change



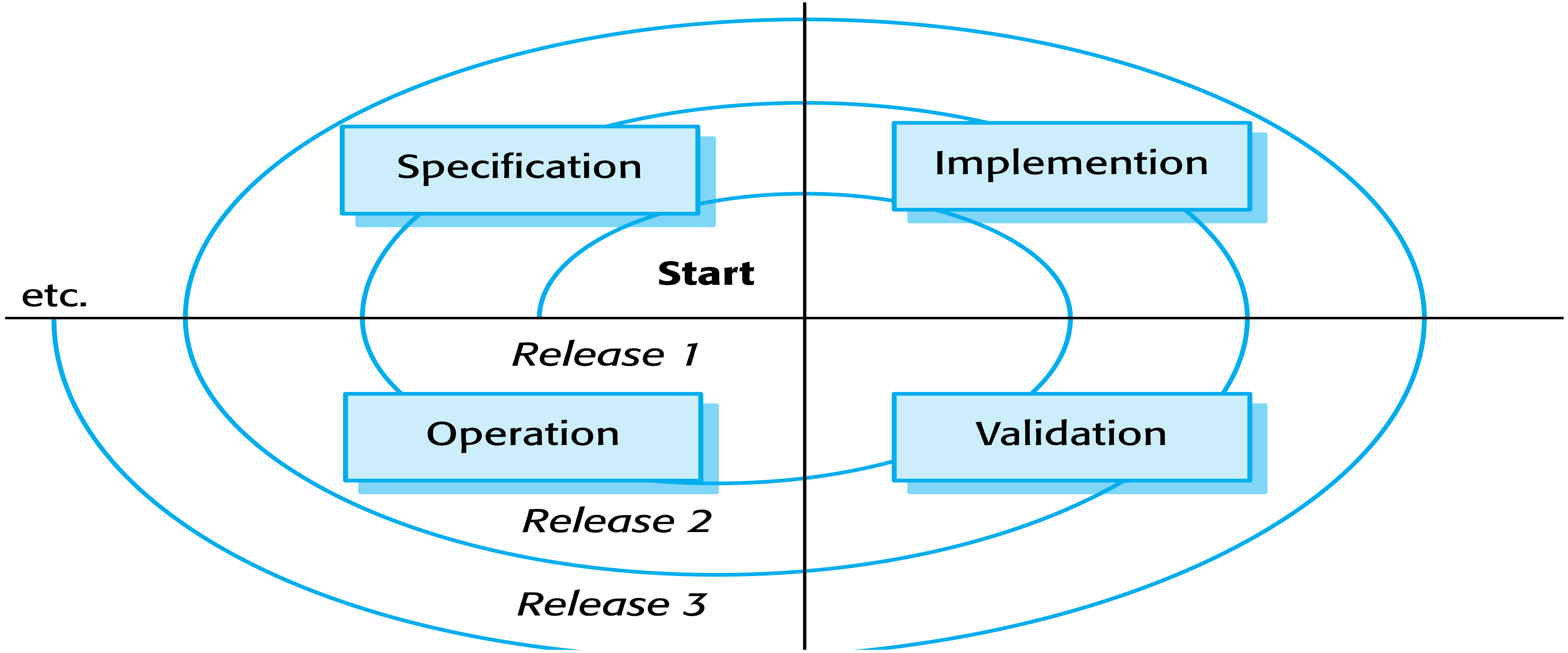
- ✧ Software change is inevitable
  - New requirements emerge when the software is used;
  - The business environment changes;
  - Errors must be repaired;
  - New computers and equipment is added to the system;
  - The performance or reliability of the system may have to be improved.
- ✧ A key problem for all organizations is implementing and managing change to their existing software systems.

Importance of evolution

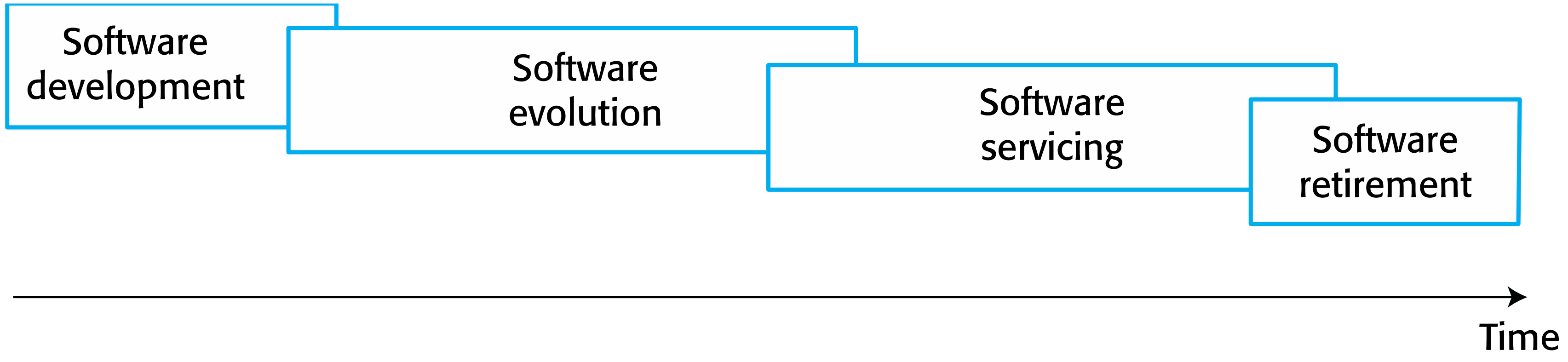


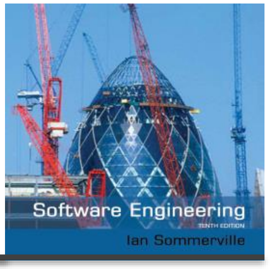
- ✧ Organisations have huge investments in their software systems - they are critical business assets.
- ✧ To maintain the value of these assets to the business, they must be changed and updated.
- ✧ The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

A spiral model of development and evolution



Evolution and servicing





Evolution and servicing

✧ Evolution

- The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.

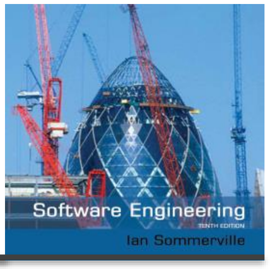
✧ Servicing

- At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in the software's environment. No new functionality is added.

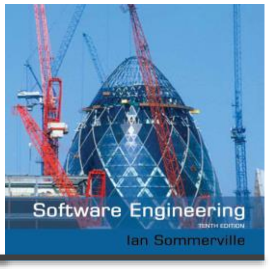
✧ Phase-out

- The software may still be used but no further changes are made to it.

Agile methods and evolution

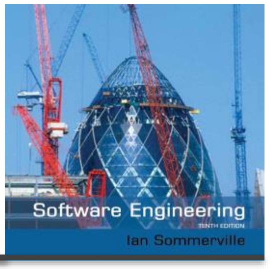


- ✧ Agile methods are based on incremental development so the transition from development to evolution is a seamless one.
  - Evolution is simply a continuation of the development process based on frequent system releases.
- ✧ Automated regression testing is particularly valuable when changes are made to a system.
- ✧ Changes may be expressed as additional user stories.



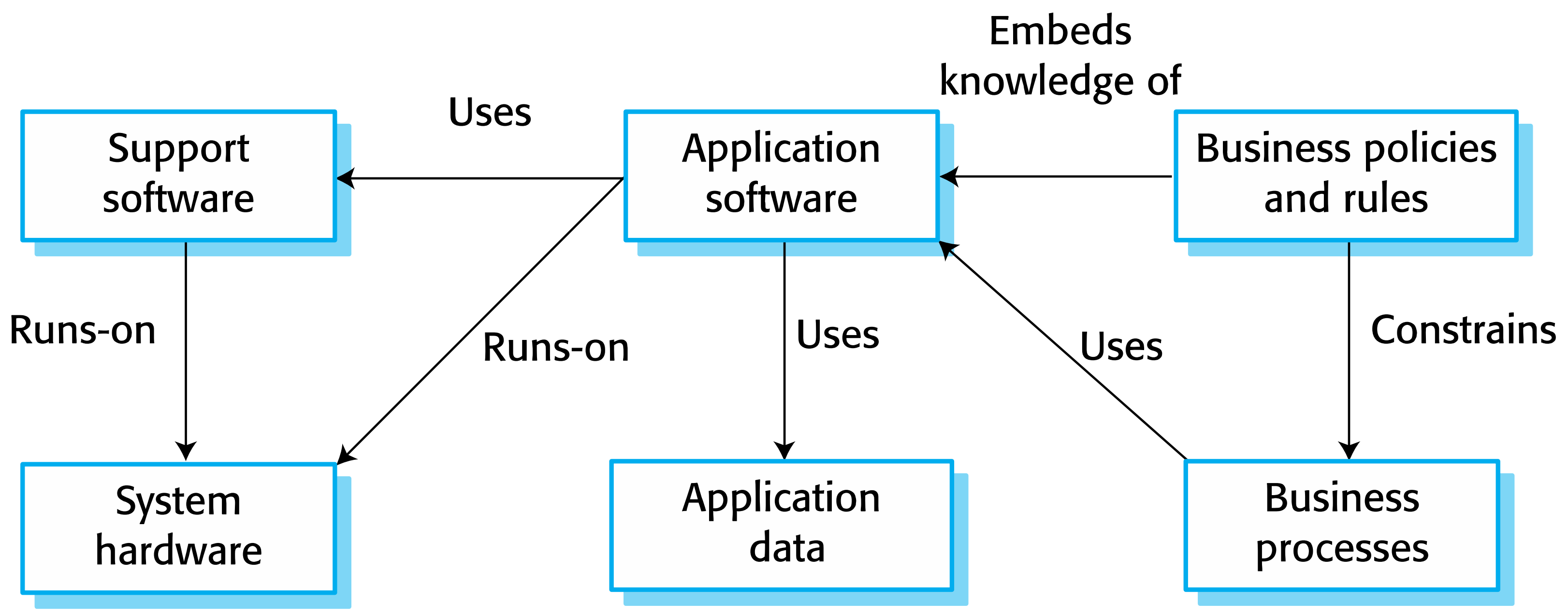
Legacy systems

Legacy systems

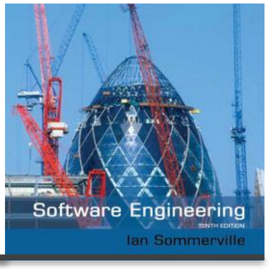


- ✧ Legacy systems are older systems that rely on languages and technology that are no longer used for new systems development.
- ✧ Legacy software may be dependent on older hardware, such as mainframe computers and may have associated legacy processes and procedures.
- ✧ Legacy systems are not just software systems but are broader socio-technical systems that include hardware, software, libraries and other supporting software and business processes.

The elements of a legacy system

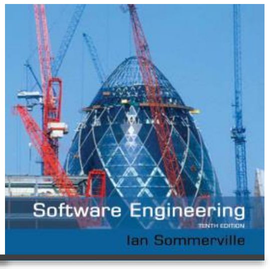


Legacy system components



- ✧ *System hardware* Legacy systems may have been written for hardware that is no longer available.
- ✧ *Support software* The legacy system may rely on a range of support software, which may be obsolete or unsupported.
- ✧ *Application software* The application system that provides the business services is usually made up of a number of application programs.
- ✧ *Application data* These are data that are processed by the application system. They may be inconsistent, duplicated or held in different databases.

Legacy system components

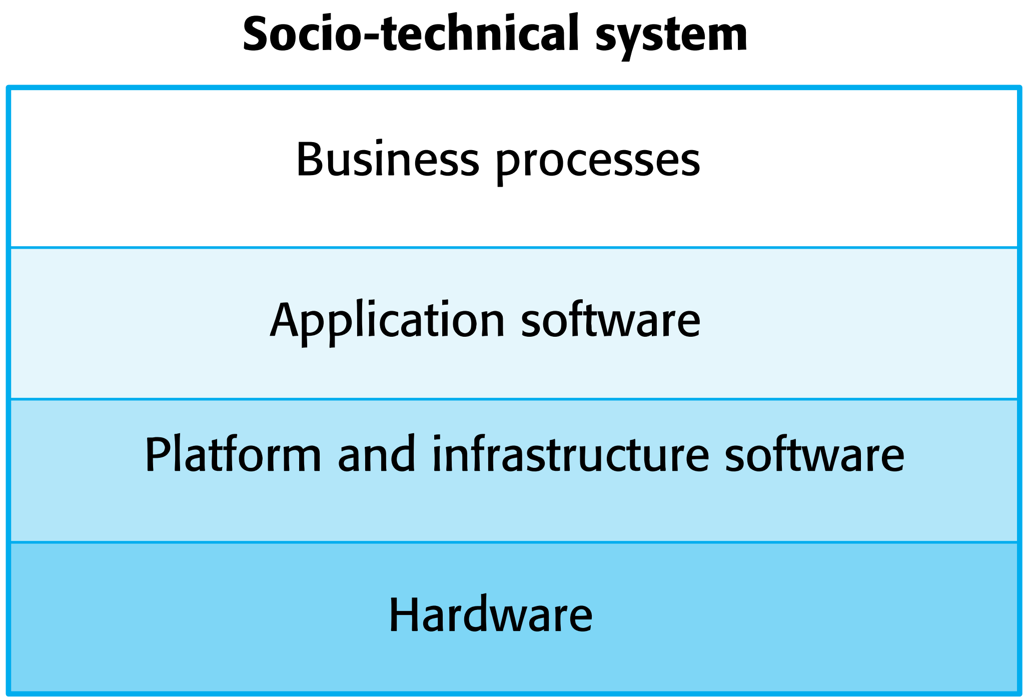


✧ *Business processes* These are processes that are used in the business to achieve some business objective.

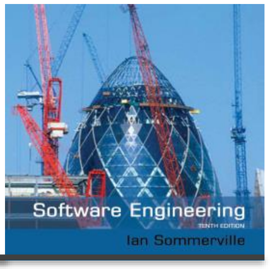
✧ Business processes may be designed around a legacy system and constrained by the functionality that it provides.

✧ *Business policies and rules* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

Legacy system layers

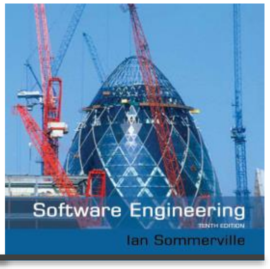


Legacy system replacement



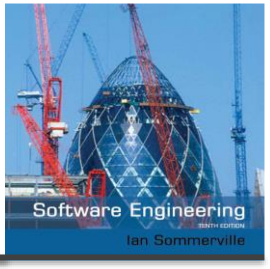
- ✧ Legacy system replacement is risky and expensive so businesses continue to use these systems
- ✧ System replacement is risky for a number of reasons
  - Lack of complete system specification
  - Tight integration of system and business processes
  - Undocumented business rules embedded in the legacy system
  - New software development may be late and/or over budget

Legacy system change



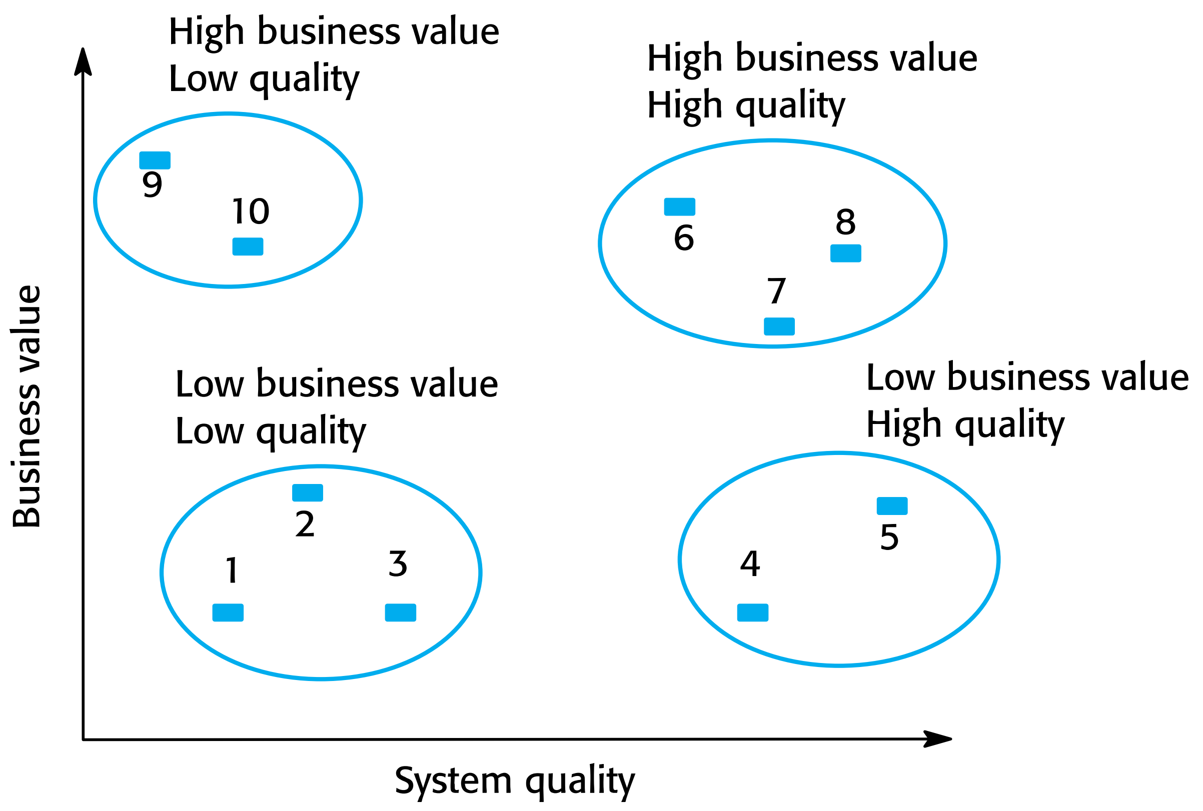
- ✧ Legacy systems are expensive to change for a number of reasons:
  - No consistent programming style
  - Use of obsolete programming languages with few people available with these language skills
  - Inadequate system documentation
  - System structure degradation
  - Program optimizations may make them hard to understand
  - Data errors, duplication and inconsistency

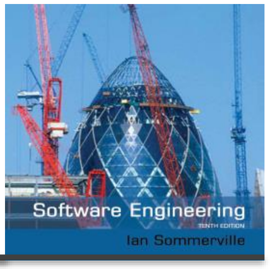
Legacy system management



- ✧ Organisations that rely on legacy systems must choose a strategy for evolving these systems
  - Scrap the system completely and modify business processes so that it is no longer required;
  - Continue maintaining the system;
  - Transform the system by re-engineering to improve its maintainability;
  - Replace the system with a new system.
- ✧ The strategy chosen should depend on the system quality and its business value.

Figure 9.13 An example of a legacy system assessment

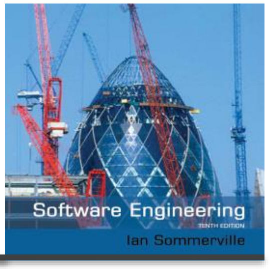




Legacy system categories

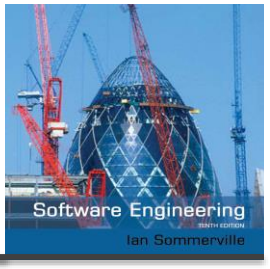
- ✧ Low quality, low business value
  - These systems should be scrapped.
- ✧ Low-quality, high-business value
  - These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available.
- ✧ High-quality, low-business value
  - Replace with COTS, scrap completely or maintain.
- ✧ High-quality, high business value
  - Continue in operation using normal system maintenance.

‘Bad smells’ in program code



- ✧ Duplicate code
  - The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
- ✧ Long methods
  - If a method is too long, it should be redesigned as a number of shorter methods.
- ✧ Switch (case) statements
  - These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.

‘Bad smells’ in program code



- ✧ Data clumping
  - Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.
- ✧ Speculative generality
  - This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.