UNIT 3

Design and Implementation:

Incrementalism in Software Development (R1-6);

Object Oriented Design using UML (T1-7.1);

Design Patterns (T1-7.2);

Achieving Quality Attributes(R2-5.5);

Writing Programs (R2-7);

Chapter 7 – Design and Implementation

Topics covered

- ✧ Object-oriented design using the UML
- ✧ Design patterns
- ✧ Implementation issues
- ✧ Open source development

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** models, 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.
- To develop a system design from concept to detailed, object-oriented design, one need to:
  1. **Define the context and modes of use of the system;**
  2. **Design the system architecture;**
  3. **Identify the principal system objects;**
  4. **Develop design models;**
  5. **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

✧ System context models and interaction models present complementary views of the relationships between a system and its environment:

1. **A system context model** is a structural model that demonstrates the other systems in the environment of the system being developed.
2. **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

https://www.youtube.com/watch?v=V0feqJiE8OY

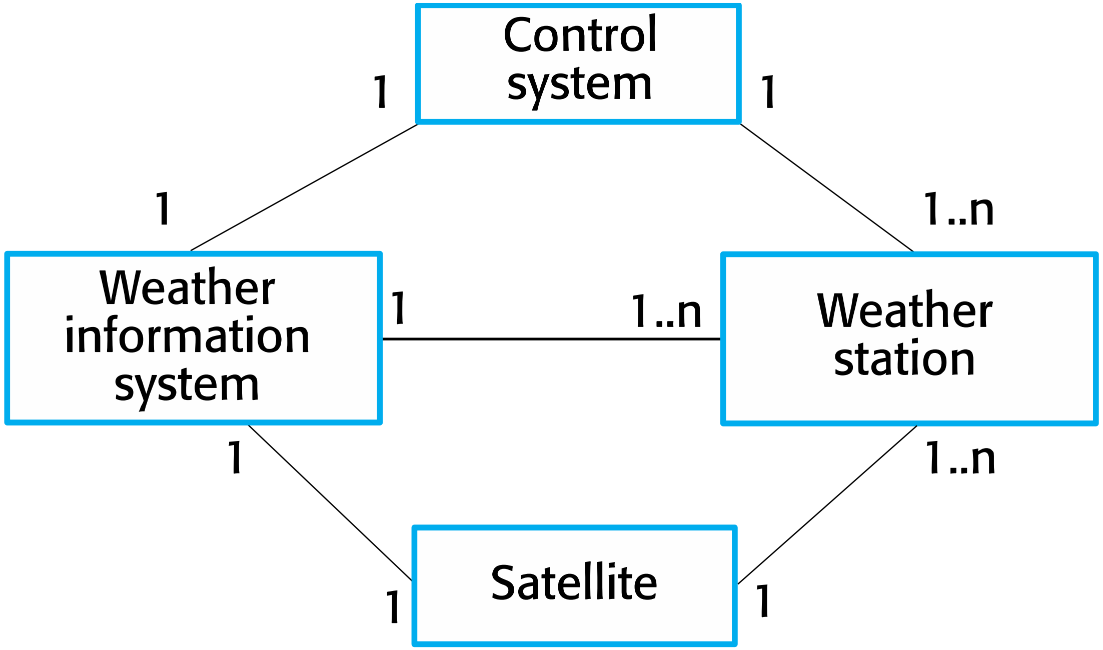
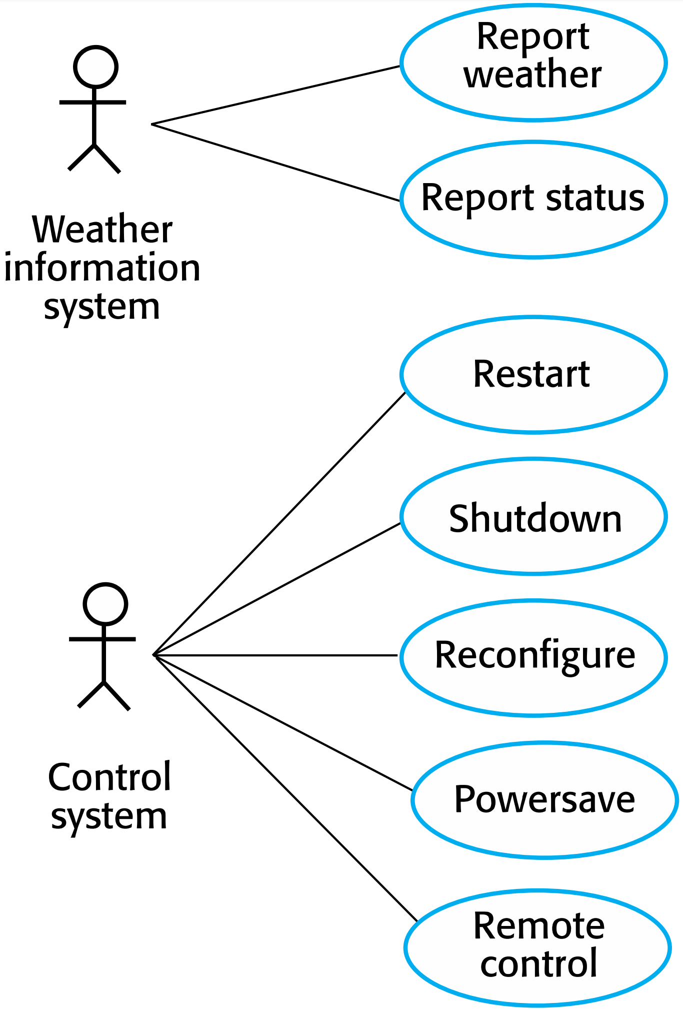


Figure shows that the systems in the environment of each weather station are a weather information system, an onboard satellite system, and a control system. The cardinality information on the link shows that there is a single control system but several weather stations, one satellite, and one general weather information system.

https://www.youtube.com/watch?v=insYyfCr25s

Weather station use cases



- Figure shows that the weather station interacts with the weather information system to report weather data and the status of the weather station hardware.
- Other interactions are with a control system that can issue specific weather station control commands.
- The stick figure is used in the UML to represent other systems as well as human users

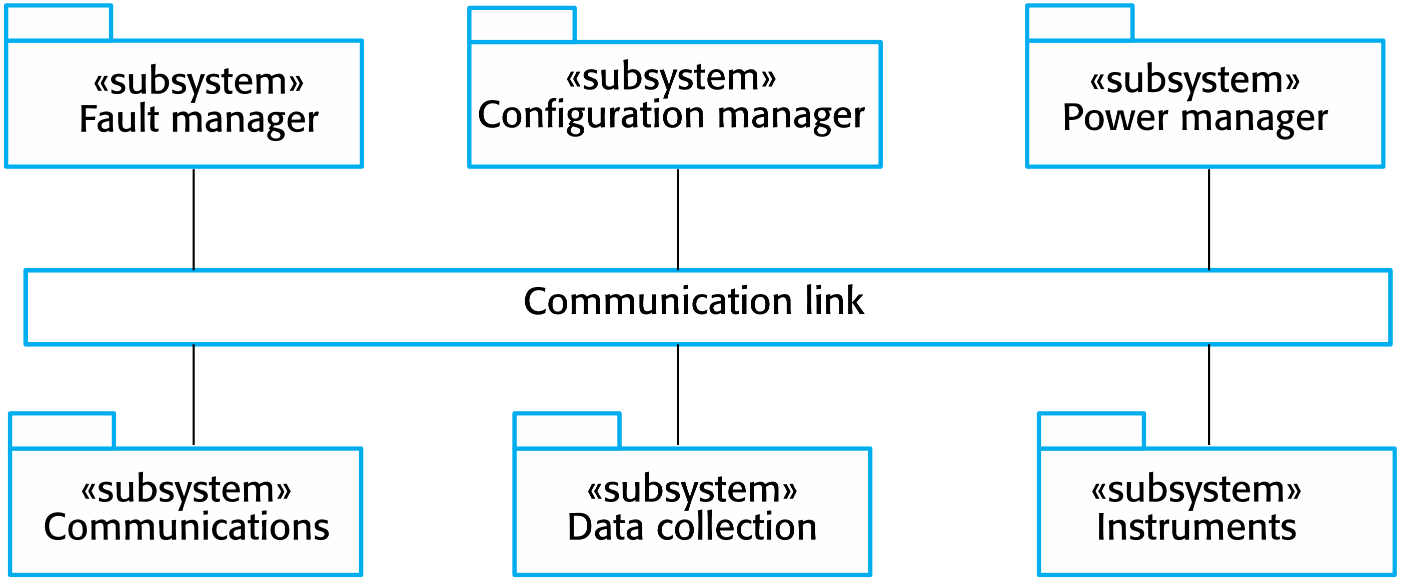
Use case description—Report weather

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

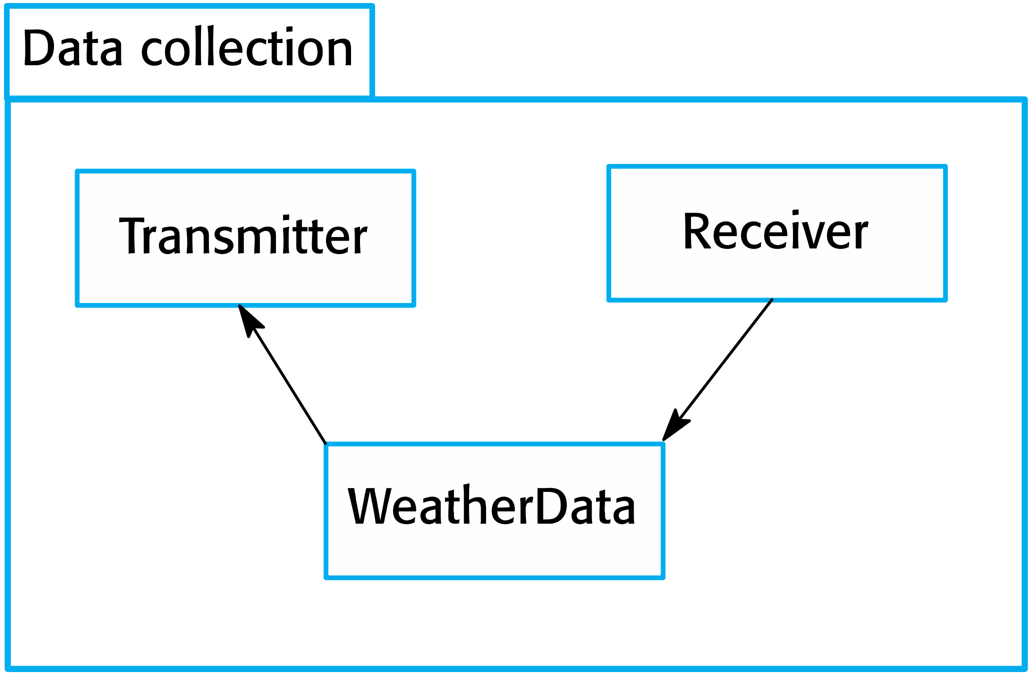
- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- You identify the major components that make up the system and their interactions and then may organize the components using an architectural pattern such 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


«subsystem» Fault manager
«subsystem» Configuration manager
«subsystem» Power manager
Communication link
«subsystem» Communications
«subsystem» Data collection
«subsystem» Instruments

- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure, shown as Communication link in Figure.
- Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them.
- This “listener model” is a commonly used architectural style for distributed systems

Architecture of data collection system



The Transmitter and Receiver objects are concerned with managing communications, and the Weather Data object encapsulates the information that is collected from the instruments and transmitted to the weather information system. This arrangement follows **the producer–consumer pattern**,

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

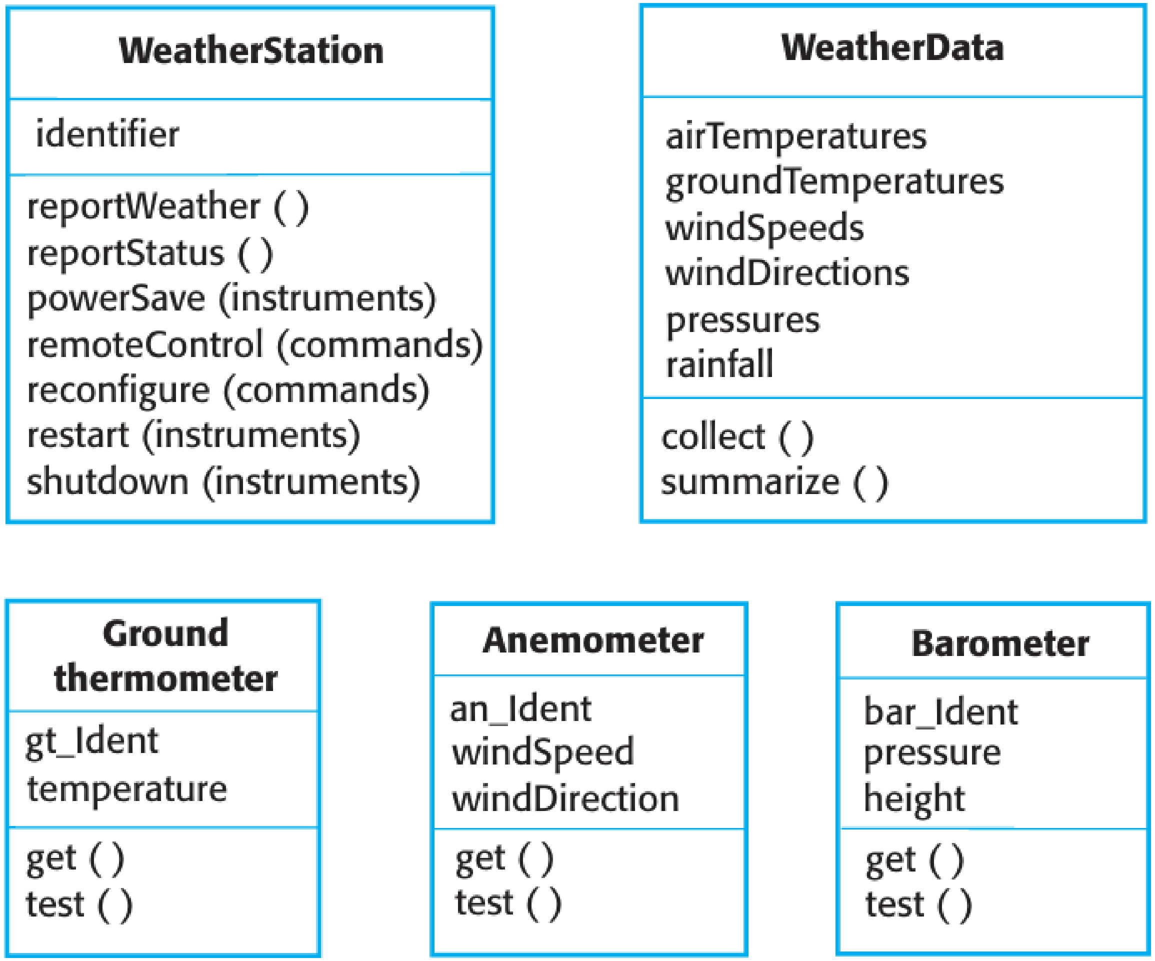
1. **Use a grammatical analysis of a natural language description of the system to be constructed.** Objects and attributes are nouns; operations or services are verbs
2. **Use tangible entities (things)** in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations such as offices, organizational units such as companies, and so on
3. **Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn**. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations

Weather station object classes

✧ Object class identification in the weather station system may be based on the tangible hardware and data in the system:

- Ground thermometer, Anemometer, Barometer
    - Application domain objects that are 'hardware' objects related to the instruments in the system.
- Weather station
    - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
- Weather data
    - Encapsulates the summarized data from the instruments.

Weather station object classes



WeatherStation

identifier

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

WeatherData

airTemperatures
groundTemperatures
windSpeeds
windDirections
pressures
rainfall

collect ( )
summarize ( )

Ground thermometer

gt_Ident
temperature

get ( )
test ( )

Anemometer

an_Ident
windSpeed
windDirection

get ( )
test ( )

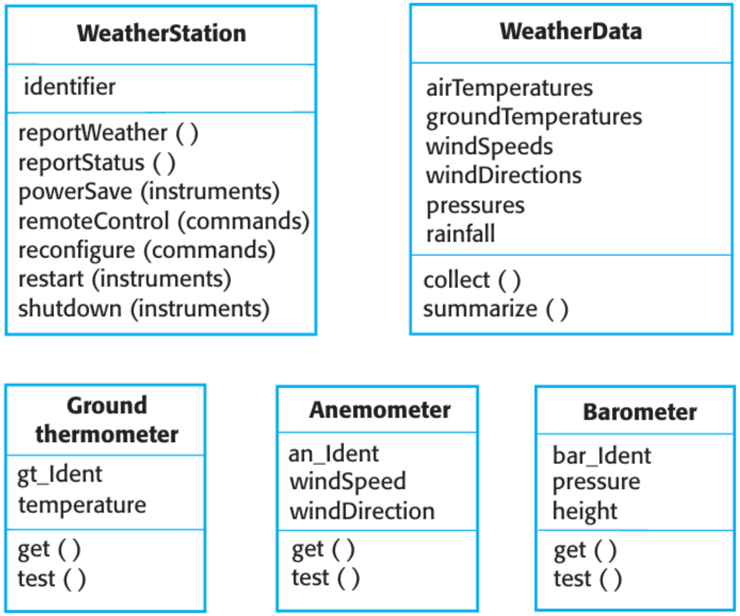
Barometer

bar_Ident
pressure
height

get ( )
test ( )

The **Ground thermometer**, **Anemometer**, and **Barometer** objects are application domain objects, and the **WeatherStation** and **WeatherData** objects have been identified from the system description and the scenario (use case) description

Weather station object classes



WeatherStation

identifier

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

WeatherData

airTemperatures
groundTemperatures
windSpeeds
windDirections
pressures
rainfall

collect ( )
summarize ( )

Ground thermometer

gt_Ident
temperature

get ( )
test ( )

Anemometer

an_Ident
windSpeed
windDirection

get ( )
test ( )

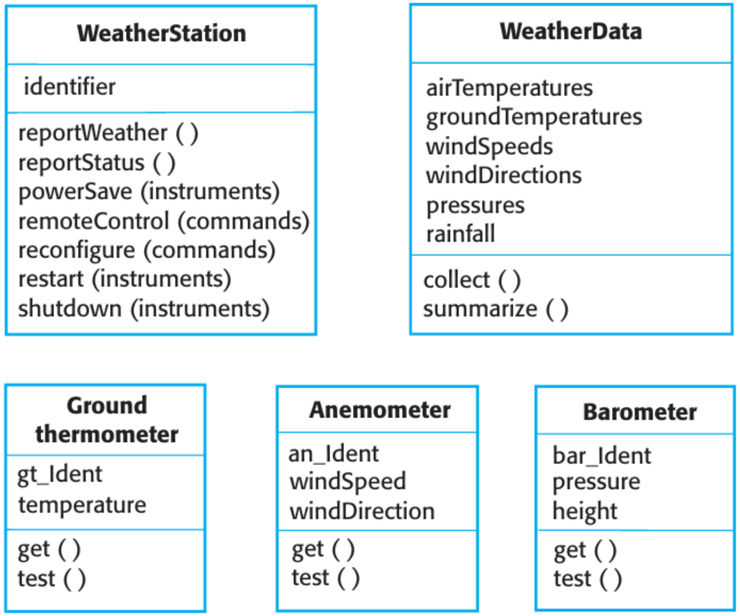
Barometer

bar_Ident
pressure
height

get ( )
test ( )

1. The WeatherStation object class provides the basic interface of the weather station with its environment. Its operations are based on the interactions.
2. The WeatherData object class is responsible for processing the report weather command. It sends the summarized data from the weather station instruments to the weather information system.
3. The Ground thermometer, Anemometer, and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware. These objects operate autonomously to collect data at the specified frequency and store the collected data locally. This data is delivered to the WeatherData object on request

Weather station object classes



WeatherStation

identifier

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

WeatherData

airTemperatures
groundTemperatures
windSpeeds
windDirections
pressures
rainfall

collect ( )
summarize ( )

Ground thermometer

gt_Ident
temperature

get ( )
test ( )

Anemometer

an_Ident
windSpeed
windDirection

get ( )
test ( )

Barometer

bar_Ident
pressure
height

get ( )
test ( )

Use knowledge of the application domain to identify other objects, attributes. and services:

1. Weather stations are often located in remote places and include various instruments that sometimes go wrong. Instrument failures should be reported automatically. This implies that you need attributes and operations to check the correct functioning of the instruments.
2. There are many remote weather stations, so each weather station should have its own identifier so that it can be uniquely identified in communications.
3. As weather stations are installed at different times, the types of instrument may be different. Therefore, each instrument should also be uniquely identified, and a database of instrument information should be maintained.

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**. Important relationships that may be documented at this stage are **generalization (inheritance) relationships, uses/used-by relationships, and composition relationships**.
  - **Dynamic models describe the dynamic interactions between objects**. Interactions that may be documented include the sequence of service requests made by objects and the state changes triggered by these object interactions

Examples of design models

1. **Subsystem models** that show logical groupings of objects into coherent subsystems.
2. **Sequence models** that show the sequence of object interactions.
3. **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.

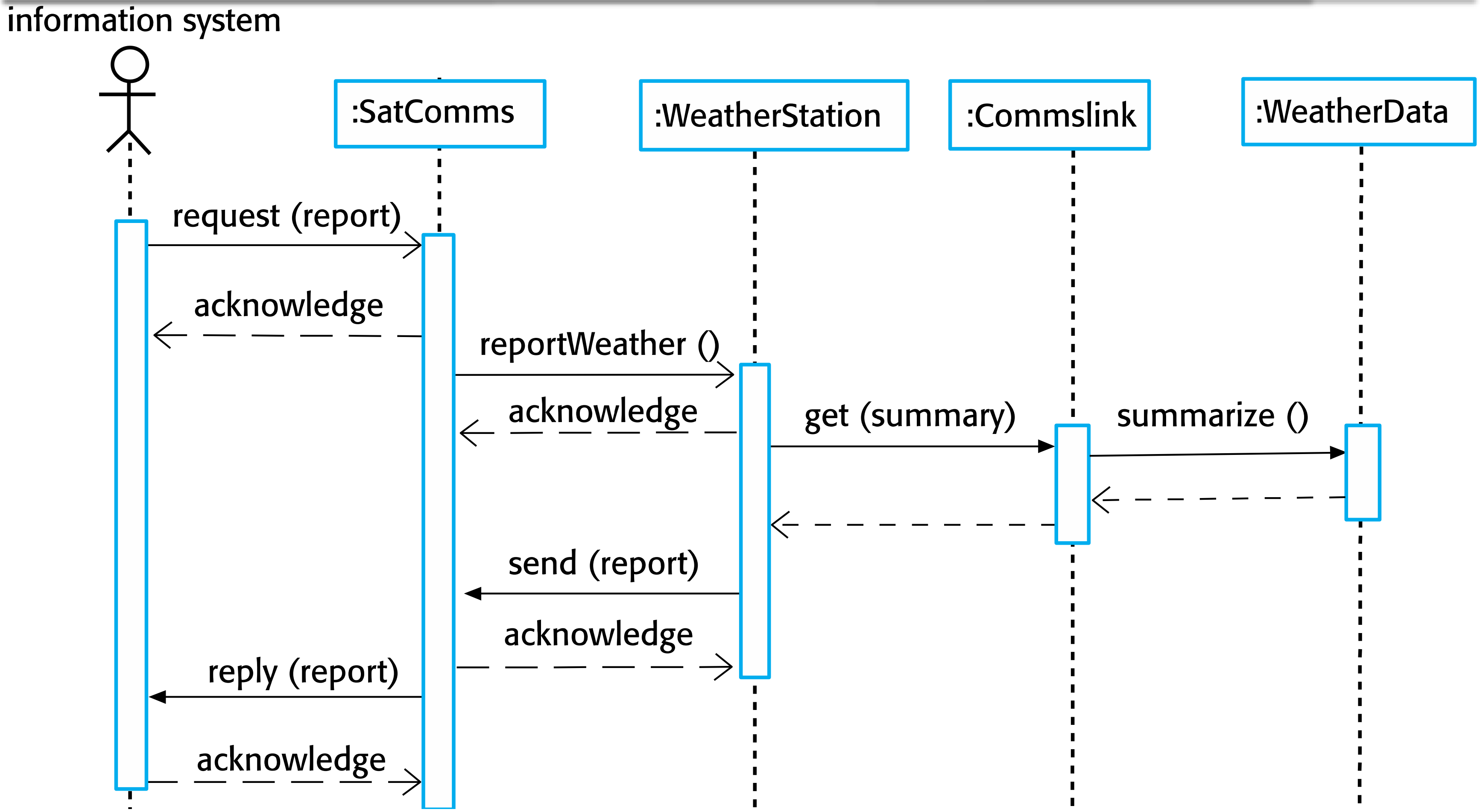
Subsystem models

- ✧ 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

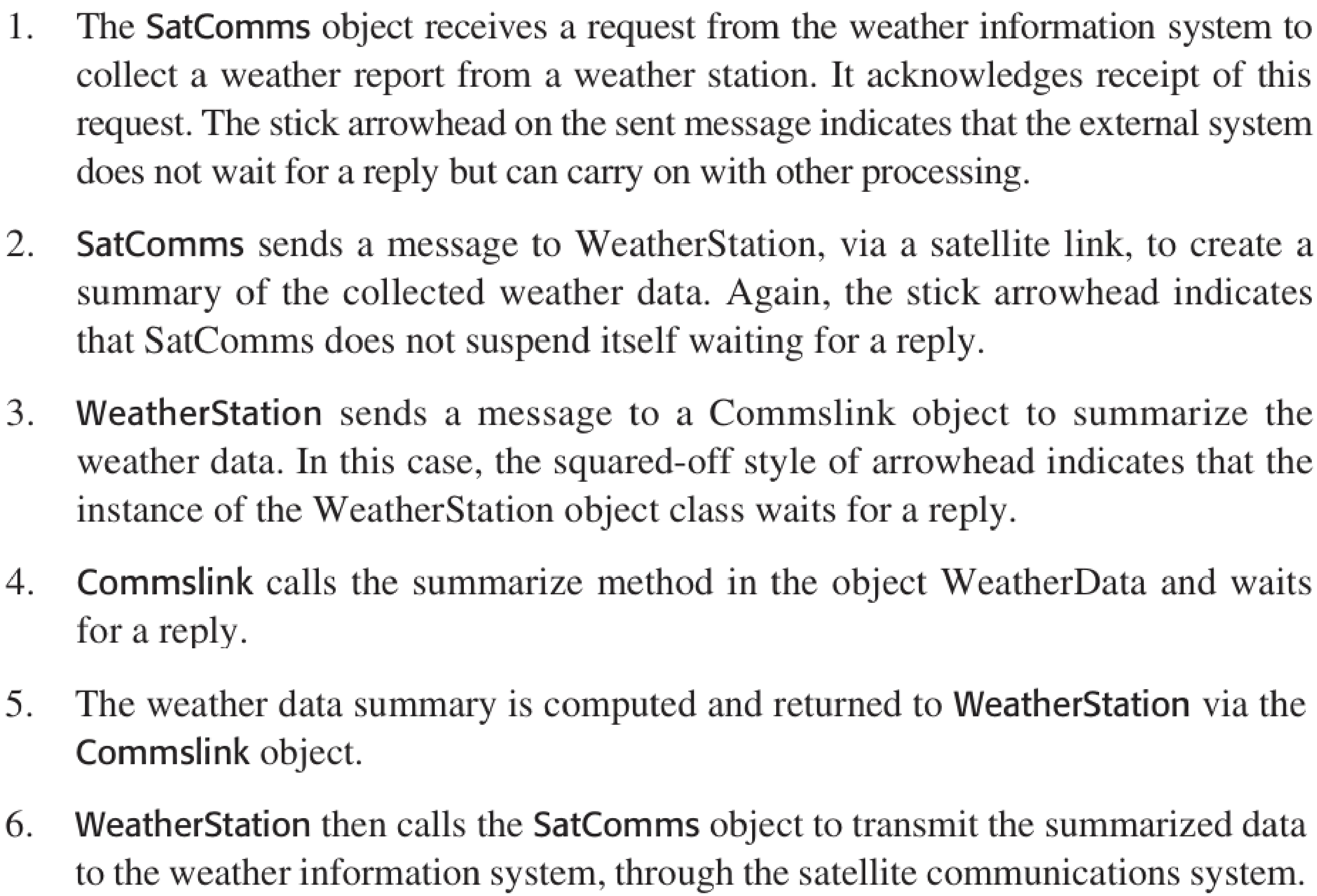
- ✧ Sequence models show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types 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



This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station.

Sequence diagram describing data collection



1. The SatComms object receives a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.

2. SatComms sends a message to WeatherStation, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.

3. WeatherStation sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.

4. Commslink calls the summarize method in the object WeatherData and waits for a reply.

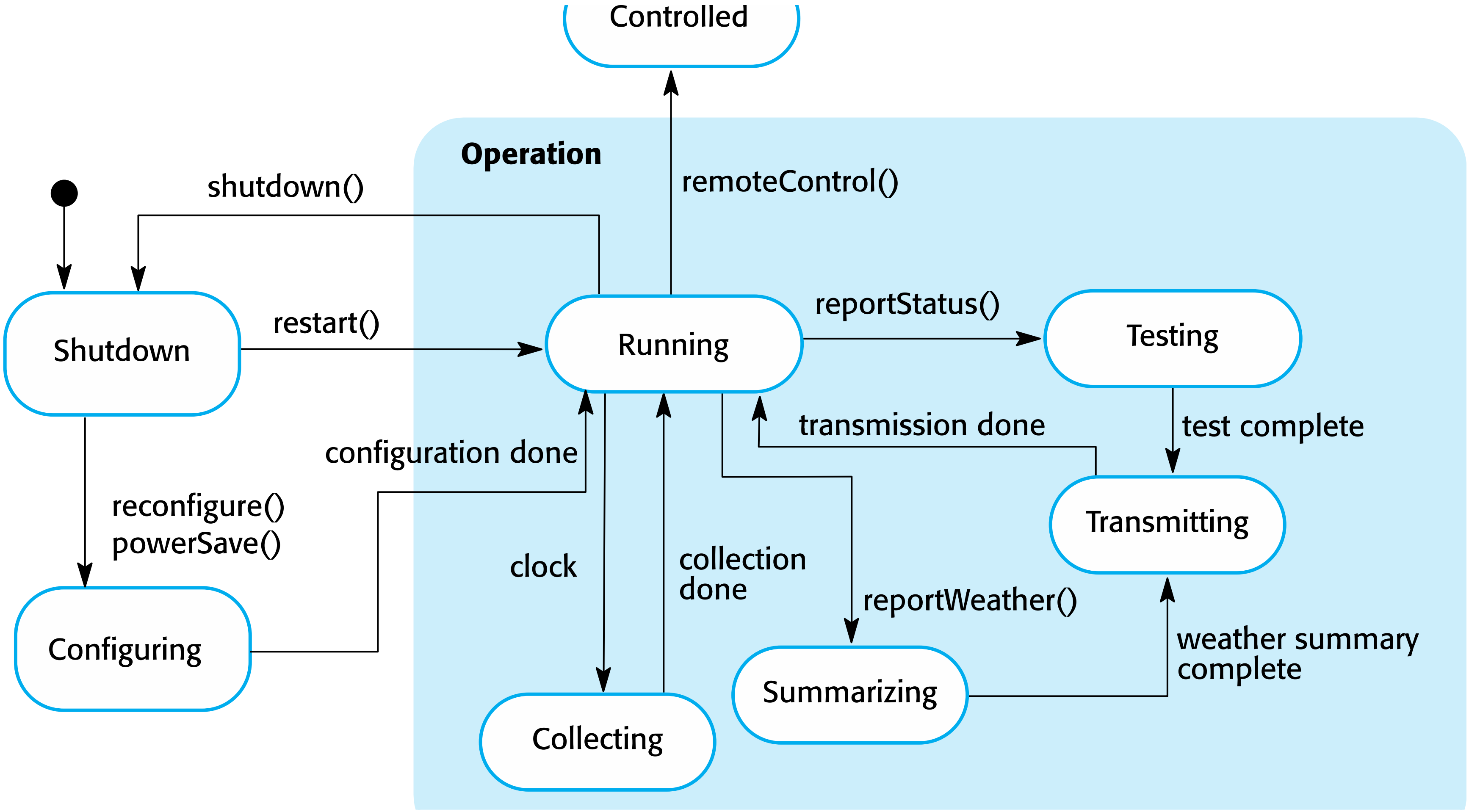
5. The weather data summary is computed and returned to WeatherStation via the Commslink object.

6. WeatherStation then calls the SatComms object to transmit the summarized data to the weather information system, through the satellite communications system.

State diagrams

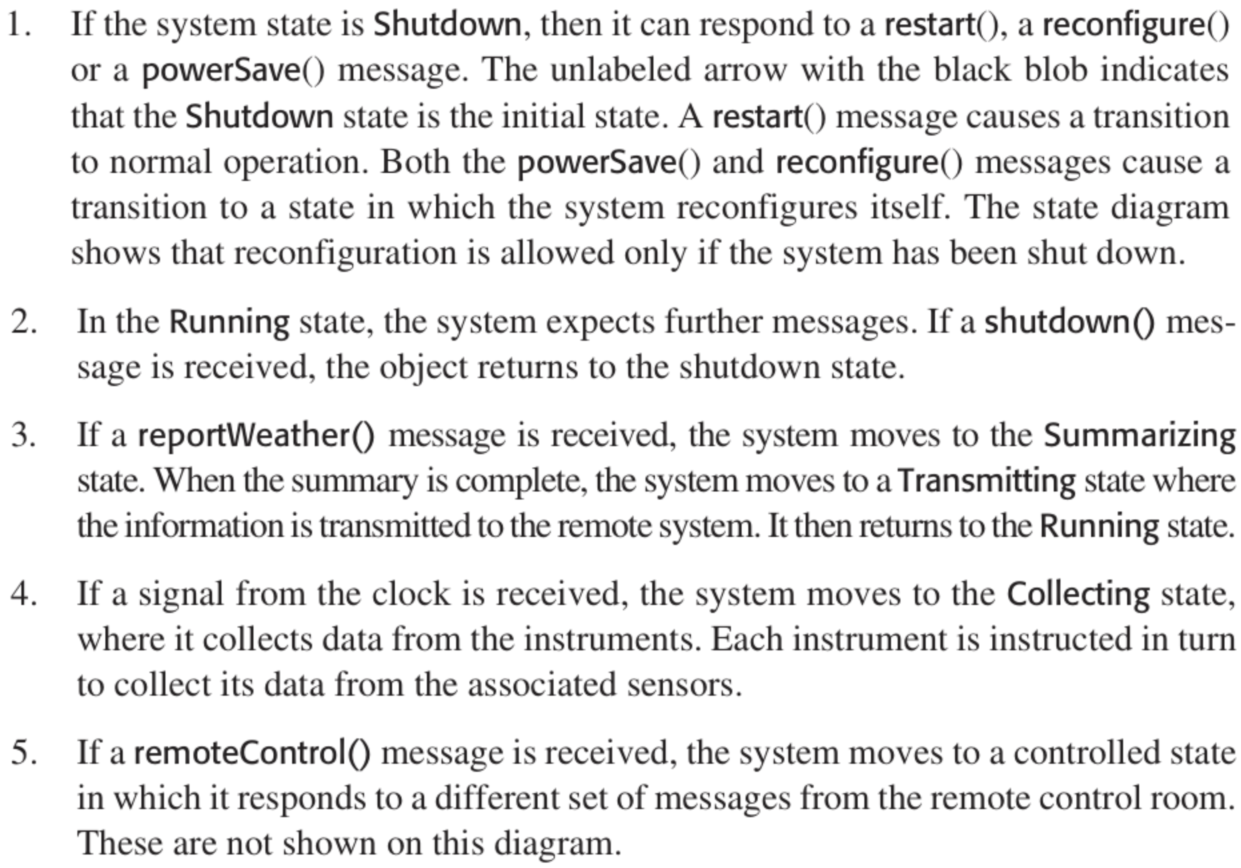
- State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- State diagrams are useful high-level models of a system or an object’s run-time behavior.
- You don’t usually need a state diagram for all of the objects in the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



shows how it responds to requests for various services.

Weather station state diagram



1. If the system state is **Shutdown**, then it can respond to a **restart**(), a **reconfigure**() or a **powerSave**() message. The unlabeled arrow with the black blob indicates that the **Shutdown** state is the initial state. A **restart**() message causes a transition to normal operation. Both the **powerSave**() and **reconfigure**() messages cause a transition to a state in which the system reconfigures itself. The state diagram shows that reconfiguration is allowed only if the system has been shut down.

2. In the **Running** state, the system expects further messages. If a **shutdown()** message is received, the object returns to the shutdown state.

3. If a **reportWeather()** message is received, the system moves to the **Summarizing** state. When the summary is complete, the system moves to a **Transmitting** state where the information is transmitted to the remote system. It then returns to the **Running** state.

4. If a signal from the clock is received, the system moves to the **Collecting** state, where it collects data from the instruments. Each instrument is instructed in turn to collect its data from the associated sensors.

5. If a **remoteControl()** message is received, the system moves to a controlled state in which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

Interface specification

- ✧ Object interfaces must 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.

Interface specification

- Interface design is concerned with specifying the detail of the interface to an object or to a group of objects.
- This means defining the signatures and semantics of the services that are provided by the object or by a group of objects.
- Interfaces can be specified in the UML using the same notation as a class diagram. However, there is no attribute section, and **the UML stereotype «interface» should be included in the name part**.
- The semantics of the interface may be defined using the object constraint language (OCL)

Interface specification

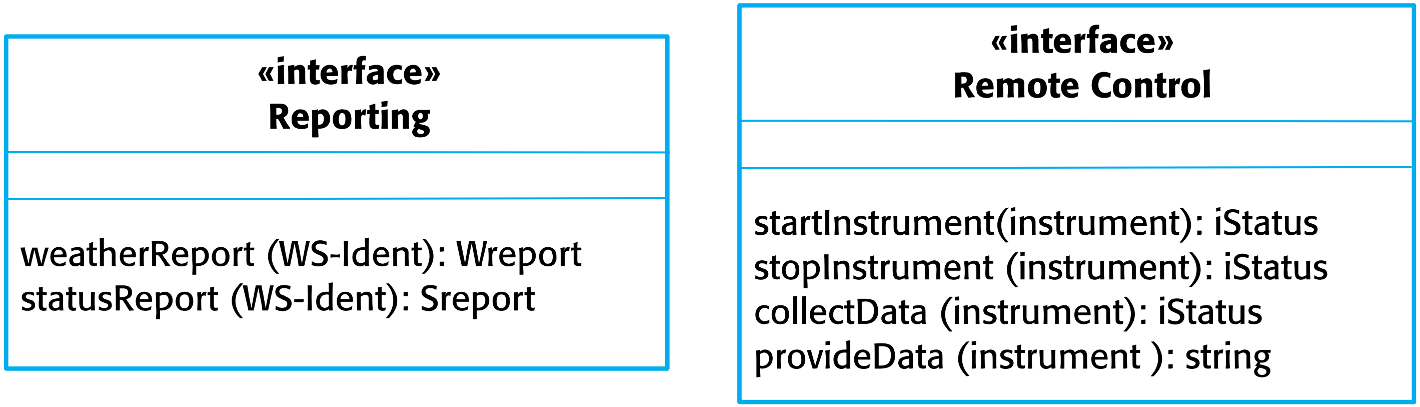
- ✧ You should not include details of the data representation in an interface design, as attributes are not defined in an interface specification. However, you should include operations to access and update data.

- ✧ As the data representation is hidden, it can be easily changed without affecting the objects that use that data. This leads to a design that is inherently more maintainable. For example, an array representation of a stack may be changed to a list representation without affecting other objects that use the stack.

- ✧ By contrast, you should normally expose the attributes in an object model, as this is the clearest way of describing the essential characteristics of the objects.

- ✧ There is not a simple 1:1 relationship between objects and interfaces. The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. This is supported directly in Java, where **interfaces are declared separately from objects and objects “implement” interfaces**. Equally, a group of objects may all be accessed through a single interface.

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

- Figure shows two interfaces that may be defined for the weather station.
- The left hand interface is a reporting interface that defines the operation names that are used to generate weather and status reports.
- These map directly to operations in the WeatherStation object.
- The remote control interface provides four operations, which map onto a single method in the WeatherStation object.
- In this case, the individual operations are encoded in the command string associated with the remoteControl method

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.

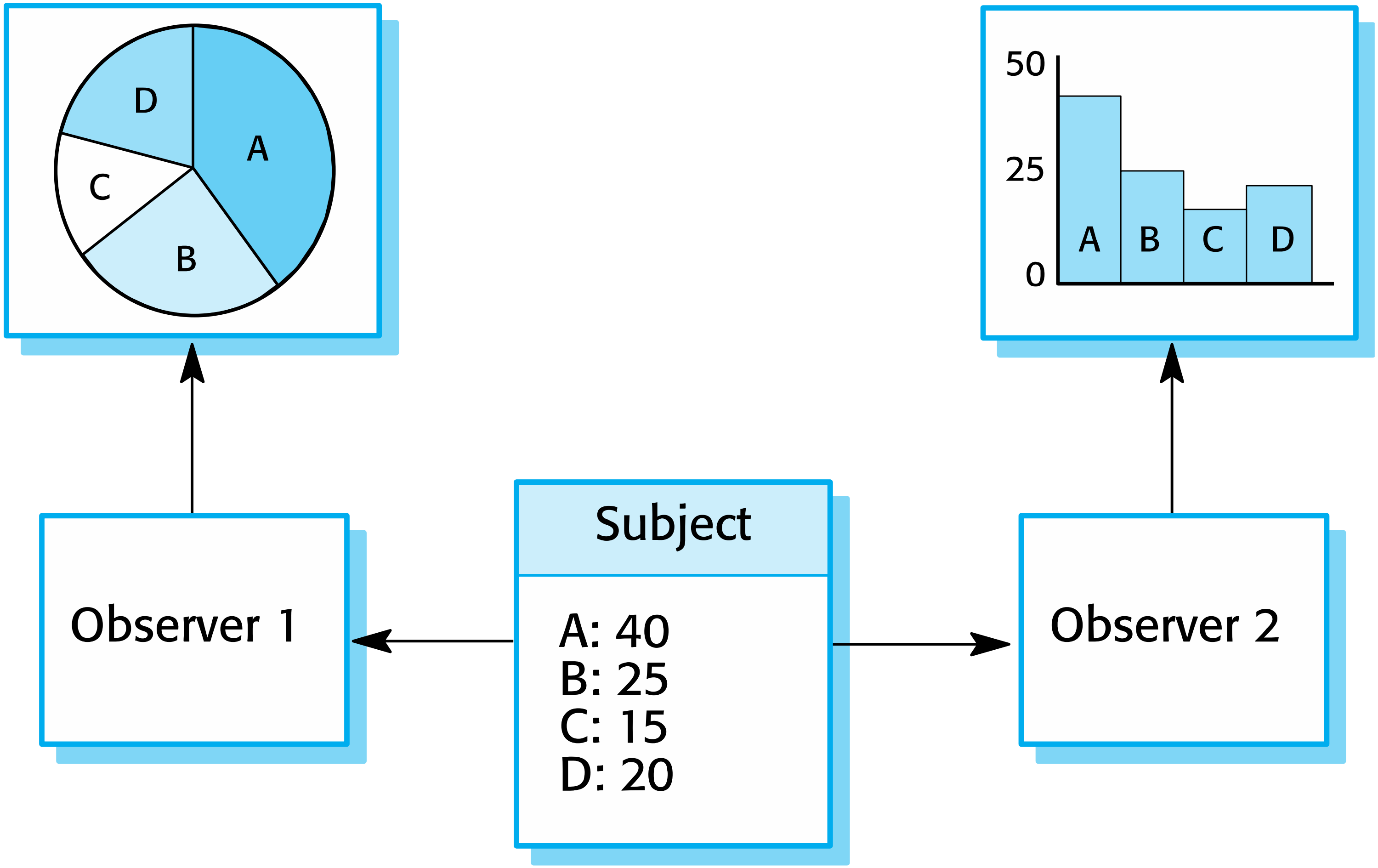
The Observer pattern

- 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.

Multiple displays using the Observer pattern



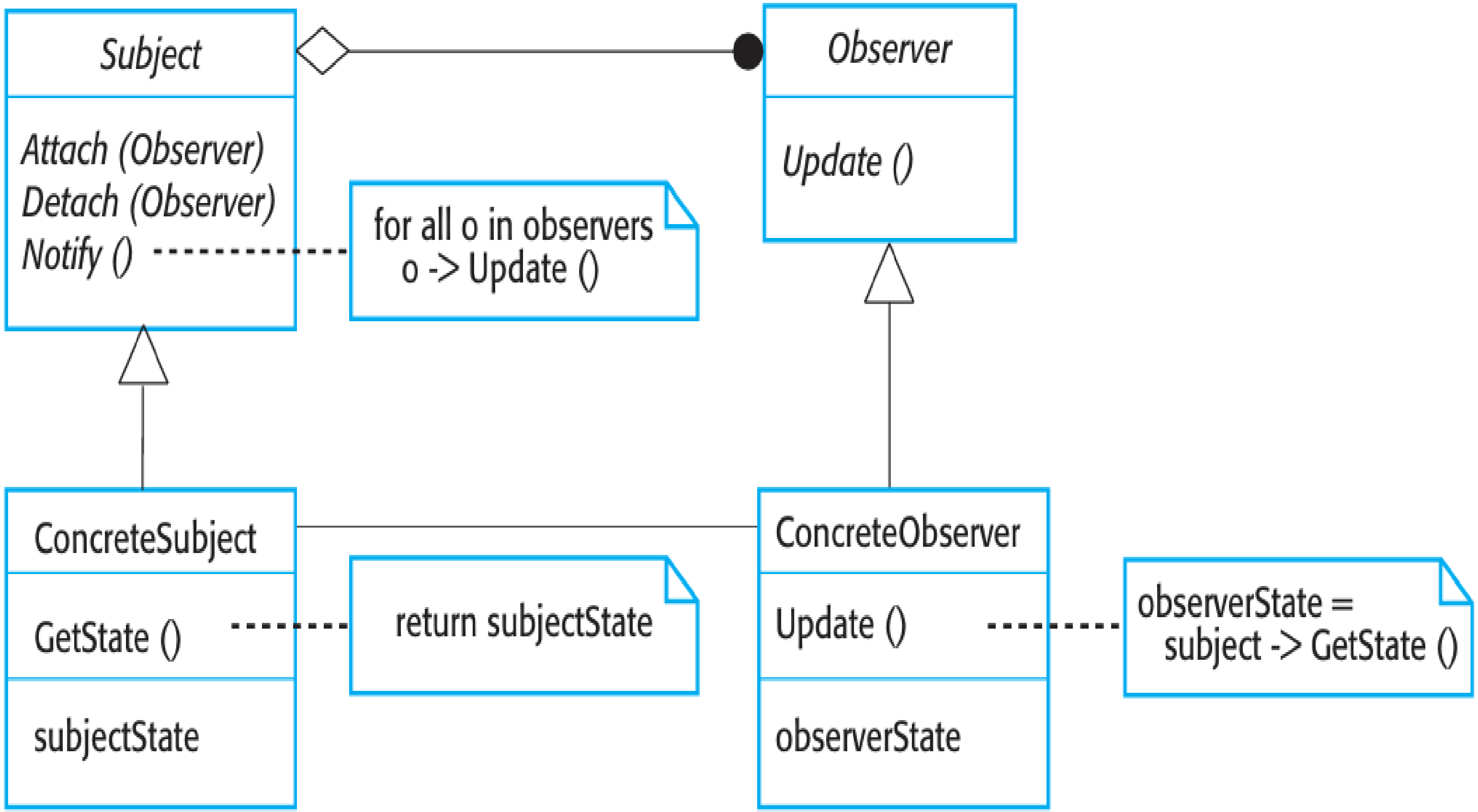
Subject

A: 40
B: 25
C: 15
D: 20

Observer 1

Observer 2

A UML model of the Observer pattern



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.

Design problems

- To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied.
  - **Tell several objects that the state of some other object has changed (Observer pattern).**
  - **Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).**
  - **Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).**
  - **Allow for the possibility of extending the functionality of an existing class at run-time (Decorator pattern).**

Implementation issues

Implementation issues

✧ Focus here is not on programming, although this is obviously important, but on other implementation issues that are often not covered in programming texts:

- Reuse Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.
- Configuration management During the development process, you have to keep track of the many different versions of each software component in a configuration management system.
- Host-target development Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system).

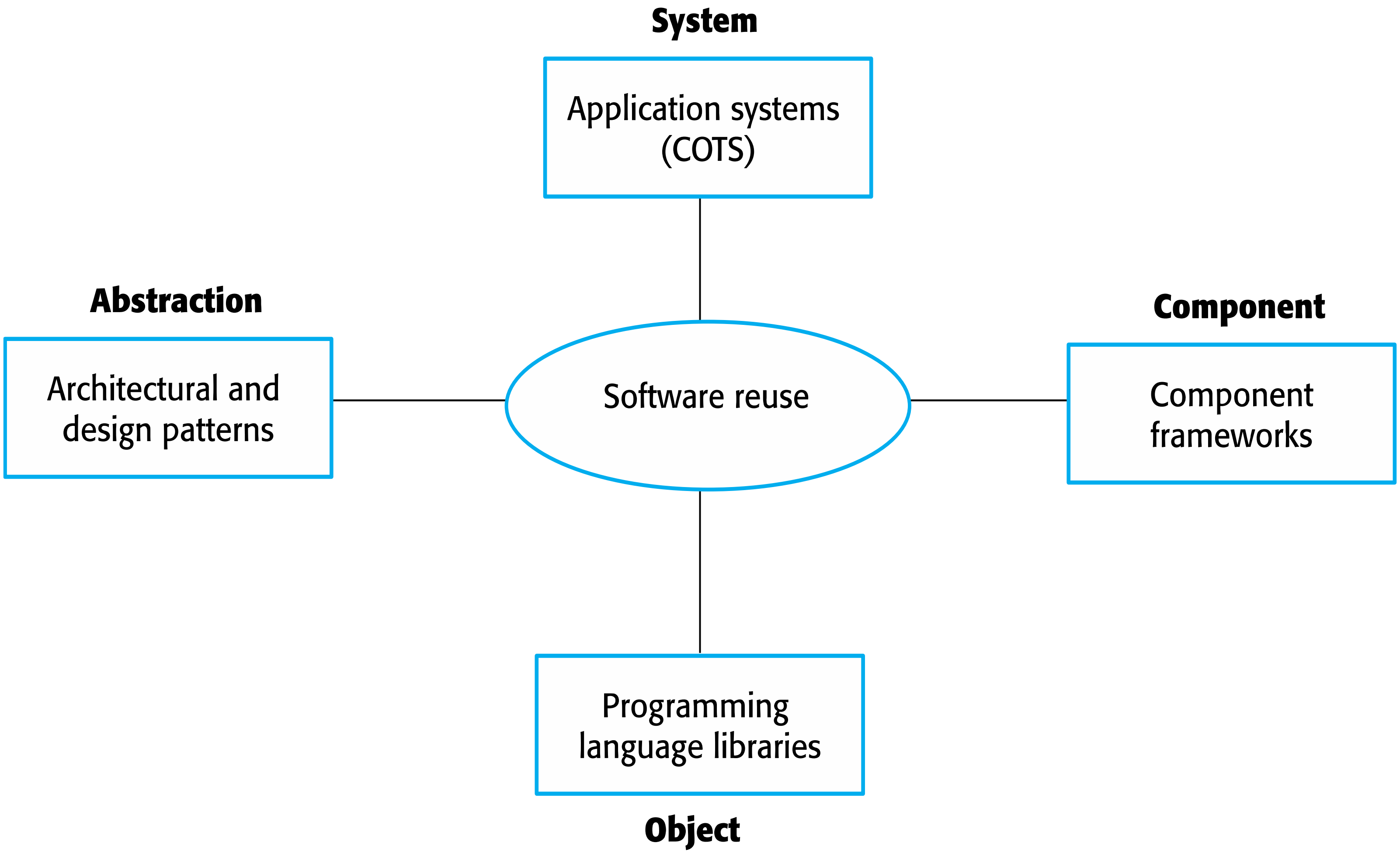
Reuse

- ✧ From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language.
  - ▪ The only significant reuse or software was the reuse of functions and objects in programming language libraries.
- ✧ Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- ✧ An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse levels

1. The abstraction level
   - At this level, you don’t reuse software directly but use knowledge of successful abstractions in the design of your software.
2. The object level
   - At this level, you directly reuse objects from a library rather than writing the code yourself.
3. The component level
   - Components are collections of objects and object classes that you reuse in application systems.
4. The system level
   - At this level, you reuse entire application systems.

Software reuse


System
Application systems (COTS)
Abstraction
Architectural and design patterns
Software reuse
Component
Component frameworks
Programming language libraries
Object

Reuse costs

1. The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
2. Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
3. The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
4. The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Configuration management

- ✧ **Configuration management is the name given to the general process of managing a changing software system**.
- ✧ The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

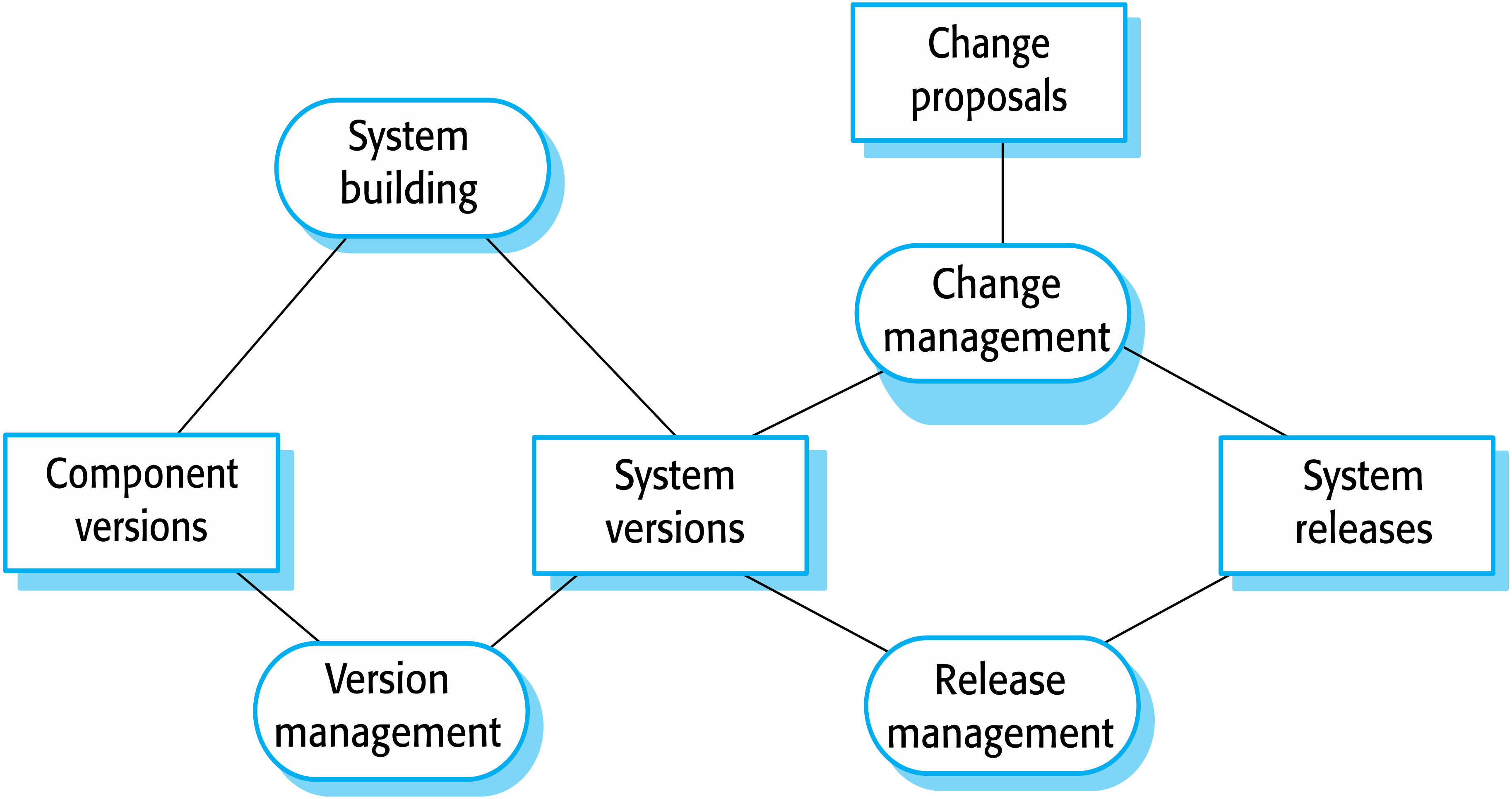
Configuration management tool interaction

- Configuration management is the name given to the general process of managing a changing software system.

- The aim of configuration management is to support the system integration process

so that all developers can access the project code and documents in a controlled way,
find out what changes have been made, and compile and link components to create a system

Configuration management tool interaction



Configuration management Activities

1. **Version management**, where support is provided to keep track of the **different versions of software components**. Version management systems include facilities to coordinate development by several programmers. They stop one developer from overwriting code that has been submitted to the system by someone else.
2. **System integration**, where support is provided **to help developers define what versions of components are used to create each version of a system description is then used to build a system automatically** by compiling and link ing the required components.
3. **Problem tracking**, where support is provided to allow **users to report bugs and other problems, and to allow all developers to see who is working** on these problems and **when they are fixed**.
4. **Release management**, where **new versions of a software system are released to customers**. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

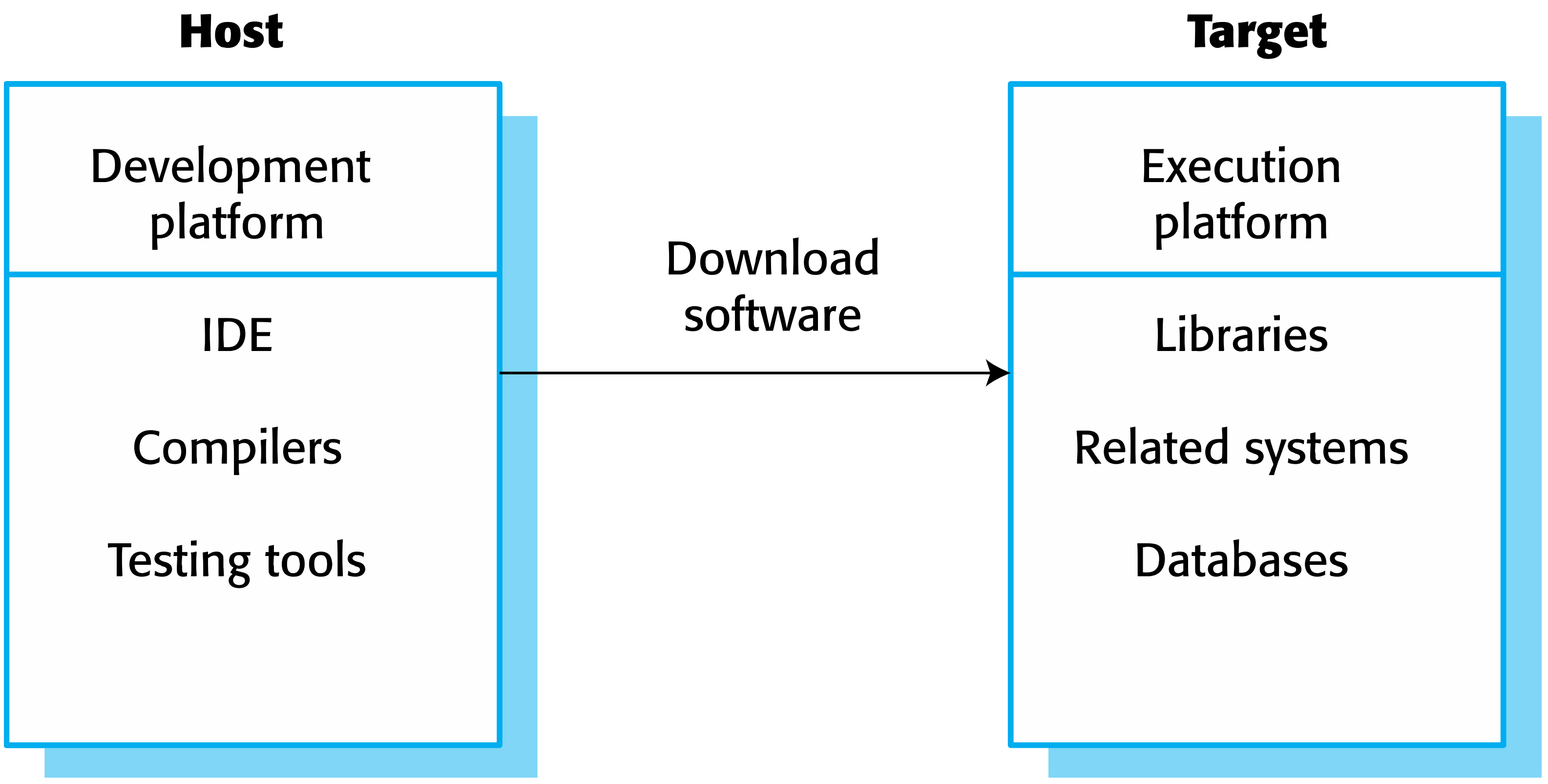
Host-target development

- Most software is developed on one computer (the host), but runs on a separate machine (the target).
- More generally, we can talk about a development platform and an execution platform.
  - A platform is more than just hardware.
  - It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures.

Host-target development

- Sometimes, the development platform and execution platform are the same, mak ing it possible to develop the software and test it on the same machine. Therefore, if you develop in Java, the target environment is the Java Virtual Machine. In principle, this is the same on every computer, so programs should be portable from one machine to another. However, particularly for embedded systems and mobile systems, the development and the execution platforms are different. You need to either move your developed software to the execution platform for testing or run a simulator on your development machine.
- Simulators are often used when developing embedded systems. You simulate hardware devices, such as sensors, and the events in the environment in which the system will be deployed. Simulators speed up the development process for embedded systems as each developer can have his or her own execution platform with no need to download the software to the target hardware. However, simulators are expensive to develop and so are usually available only for the most popular hardware architectures. If the target system has installed middleware or other software that you need to use, then you need to be able to test the system using that software.
- It may be impractical to install that software on your development machine, even if it is the same as the target platform, because of license restrictions. If this is the case, you need to transfer your developed code to the execution platform to test the system

Host-target development



Development platform tools

- ✧ A software development platform should provide a range of tools to support soft ware engineering processes. These may include:

1. **An integrated compiler and syntax-directed editing system that allows you to create, edit, and compile code.**
2. **A language debugging system.**
3. **Graphical editing tools, such as tools to edit UML models.**
4. **Testing tools, such as JUnit, that can automatically run a set of tests on a new version of a program.**
5. **Tools to support refactoring and program visualization.**
6. **Configuration management tools to manage source code versions and to integrate and build systems**

Integrated development environments (IDEs)

- ✧ Software development tools are often grouped to create an integrated development environment (IDE).
- ✧ An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- ✧ IDEs are created to support development in a specific programming language such as Java. The language IDE may be developed specially, or may be an instantiation of a general-purpose IDE, with specific language-support tools.

Component/system deployment factors

- If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.
- High availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.
- If there is a high level of communications traffic between components, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other. This reduces the delay between the time a message is sent by one component and received by another.

Component/system deployment factors

- ✧ As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform. This is straightforward for embedded systems, where the target is usually a single computer. However, for distributed systems, you need to decide on the specific platforms where the components will be deployed. Issues that you have to consider in making this decision are:

1. **The hardware and software requirements of a component** If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.

2. **The availability requirements of the system High-availability systems** may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.

3. **Component communications** If there is a lot of intercomponent communication, it is usually best to deploy them on the same platform or on platforms that are physically close to one another. This reduces communications latency—the delay between the time that a message is sent by one component and received by another. You can document your decisions on hardware and software deployment using UML deployment diagrams, which show how software components are distributed across hardware platforms.

Open source development

Open source development

- ✧ Open source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process
- ✧ Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- ✧ Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open source systems

- The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment.
- Other important open source products are Java, the Apache web server and the mySQL database management system.

Open source issues

1. Should the product that is being developed make use of open source components?
2. Should an open source approach be used for the software’s development?

Open source business

- More and more product companies are using an open source approach to development.
- Their business model is not reliant on selling a software product but on selling support for that product.
- They believe that involving the open source community will allow software to be developed more cheaply, more quickly and will create a community of users for the software.

Open source licensing

- ✧ A fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code.
  - ▪ Legally, the developer of the code (either a company or an individual) still owns the code. They can place restrictions on how it is used by including legally binding conditions in an open source software license.
  - ▪ Some open source developers believe that if an open source component is used to develop a new system, then that system should also be open source.
  - ▪ Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed source systems.

License models

1. The GNU General Public License (GPL). This is a so-called ‘reciprocal’ license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
2. The GNU Lesser General Public License (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
3. The Berkley Standard Distribution (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

License management

1. Establish a system for maintaining information about open-source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used. Licenses may change, so you need to know the conditions that you have agreed to.

2. Be aware of the different types of licenses and understand how a component is licensed before it is used. You may decide to use a component in one system but not in another because you plan to use these systems in different ways.

3. Be aware of evolution pathways for components. You need to know a bit about the open-source project where components are developed to understand how they might change in future.

4. Educate people about open source. It’s not enough to have procedures in place to ensure compliance with license conditions. You also need to educate devel opers about open source and open-source licensing.

5. Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this.

6. Participate in the open-source community. If you rely on open-source products, you should participate in the community and help support their development

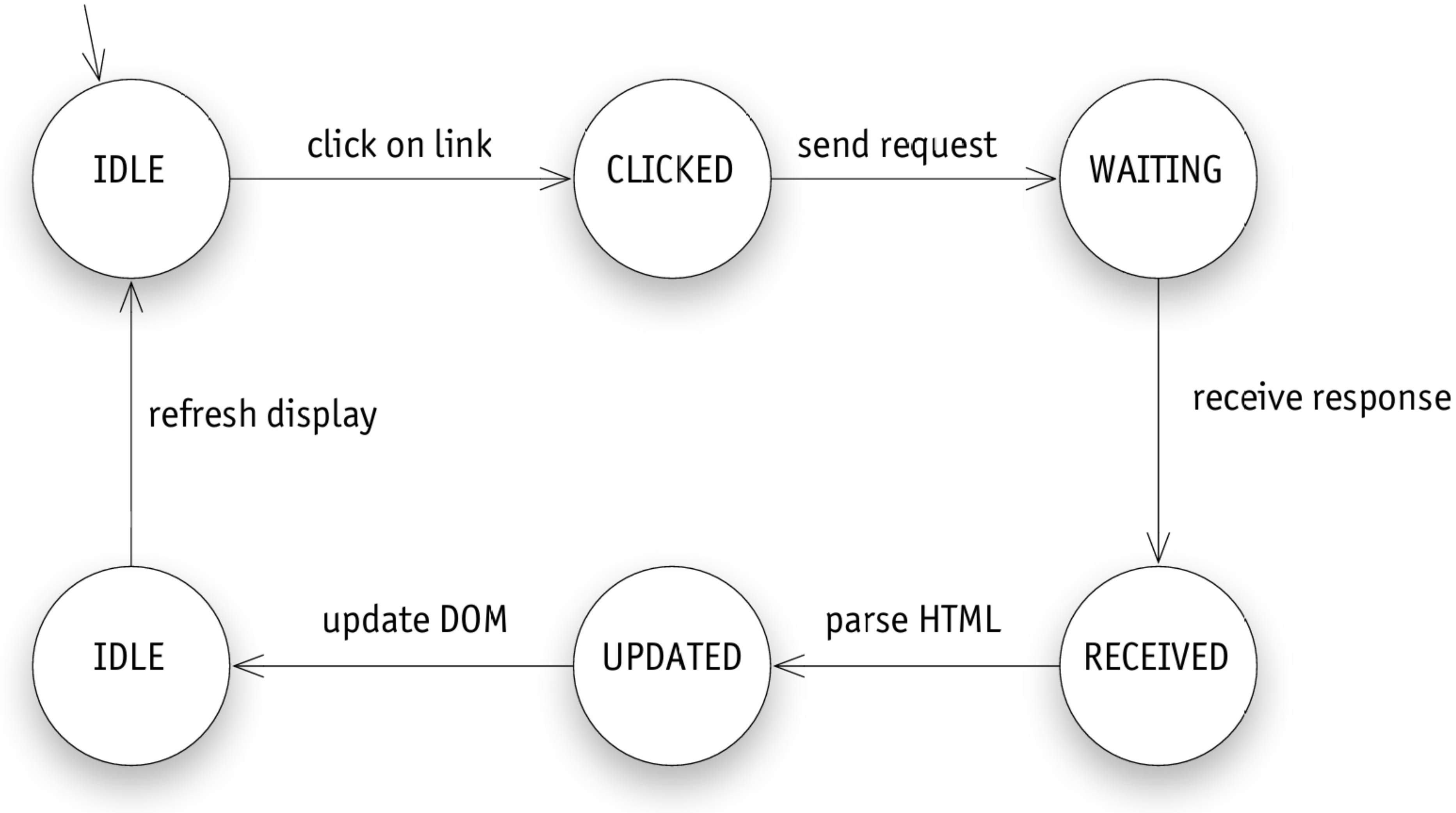
Key points

- Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- Component interfaces must be defined precisely so that other objects can use them. A UML interface stereotype may be used to define interfaces.

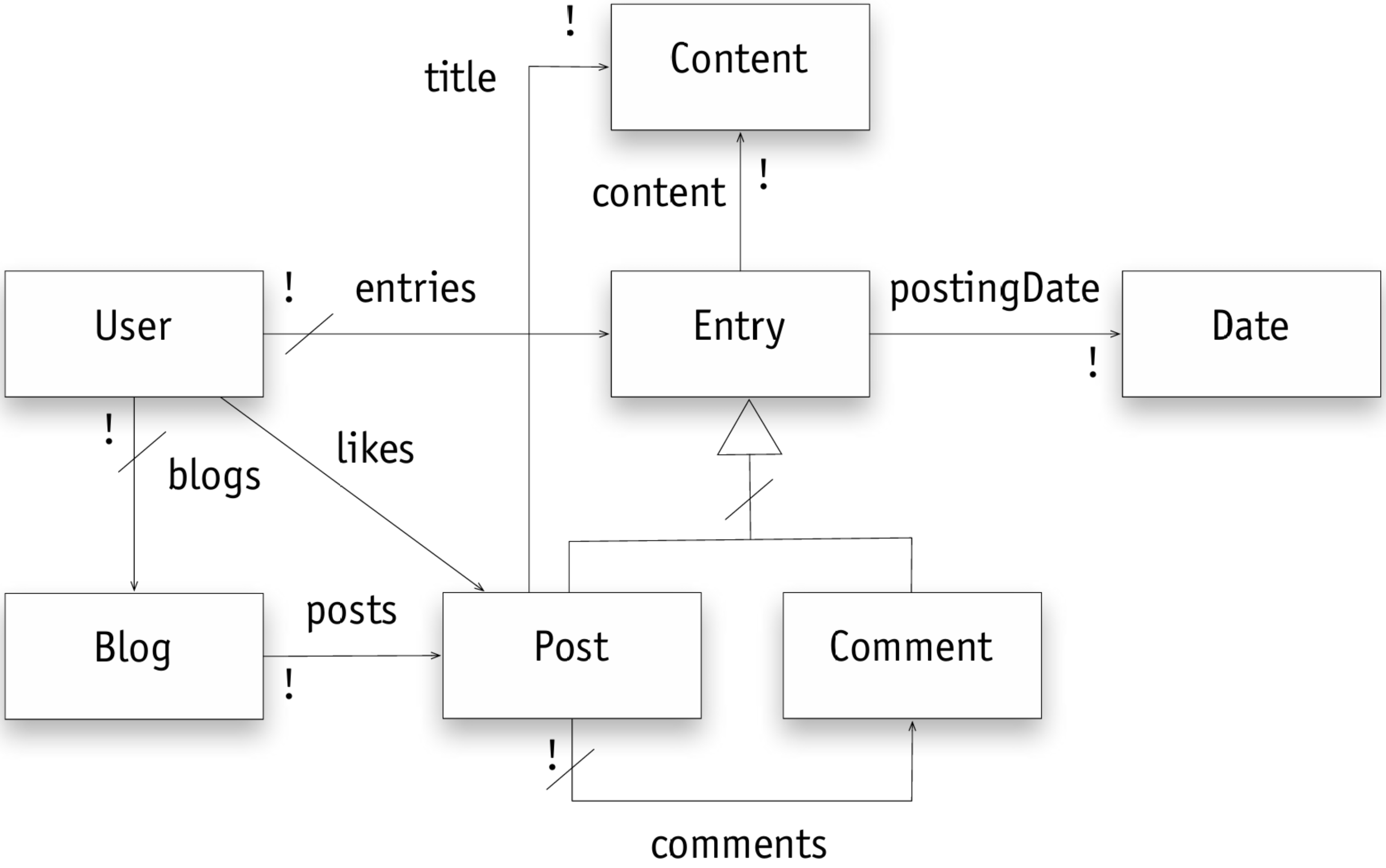
Key points

- When developing software, you should always consider the possibility of reusing existing software, either as components, services or complete systems.
- Configuration management is the process of managing changes to an evolving software system. It is essential when a team of people are cooperating to develop software.
- Most software development is host-target development. You use an IDE on a host machine to develop the software, which is transferred to a target machine for execution.
- Open source development involves making the source code of a system publicly available. This means that many people can propose changes and improvements to the software.

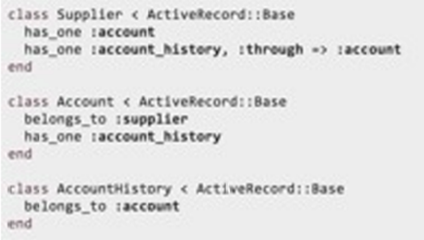
what browser does



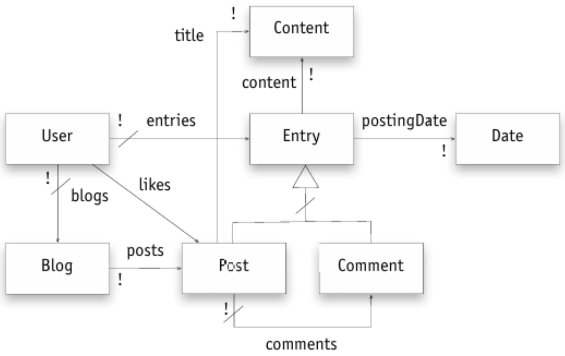
an object model



one model, many implementations



```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```



?

```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

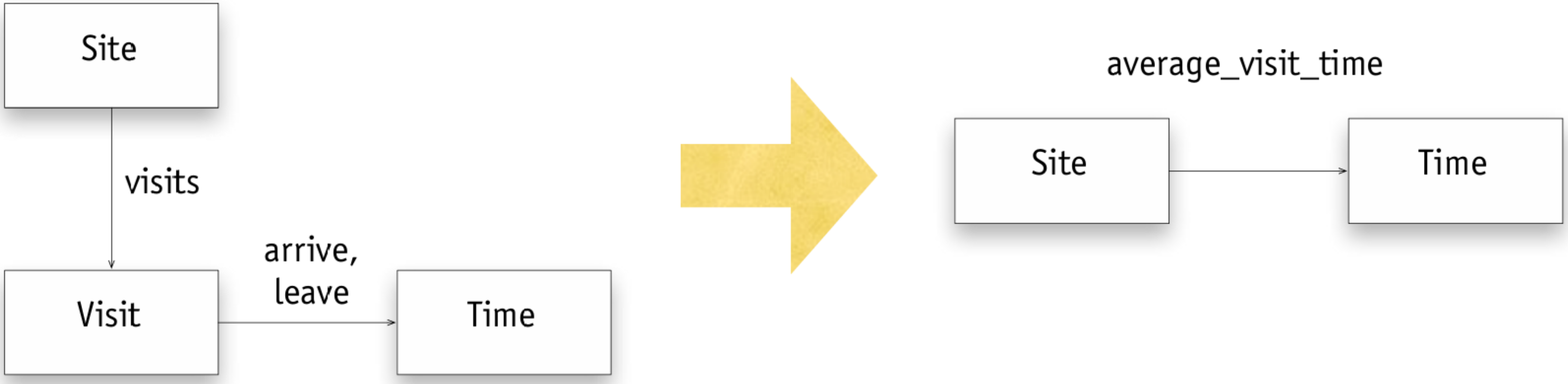
```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

principle

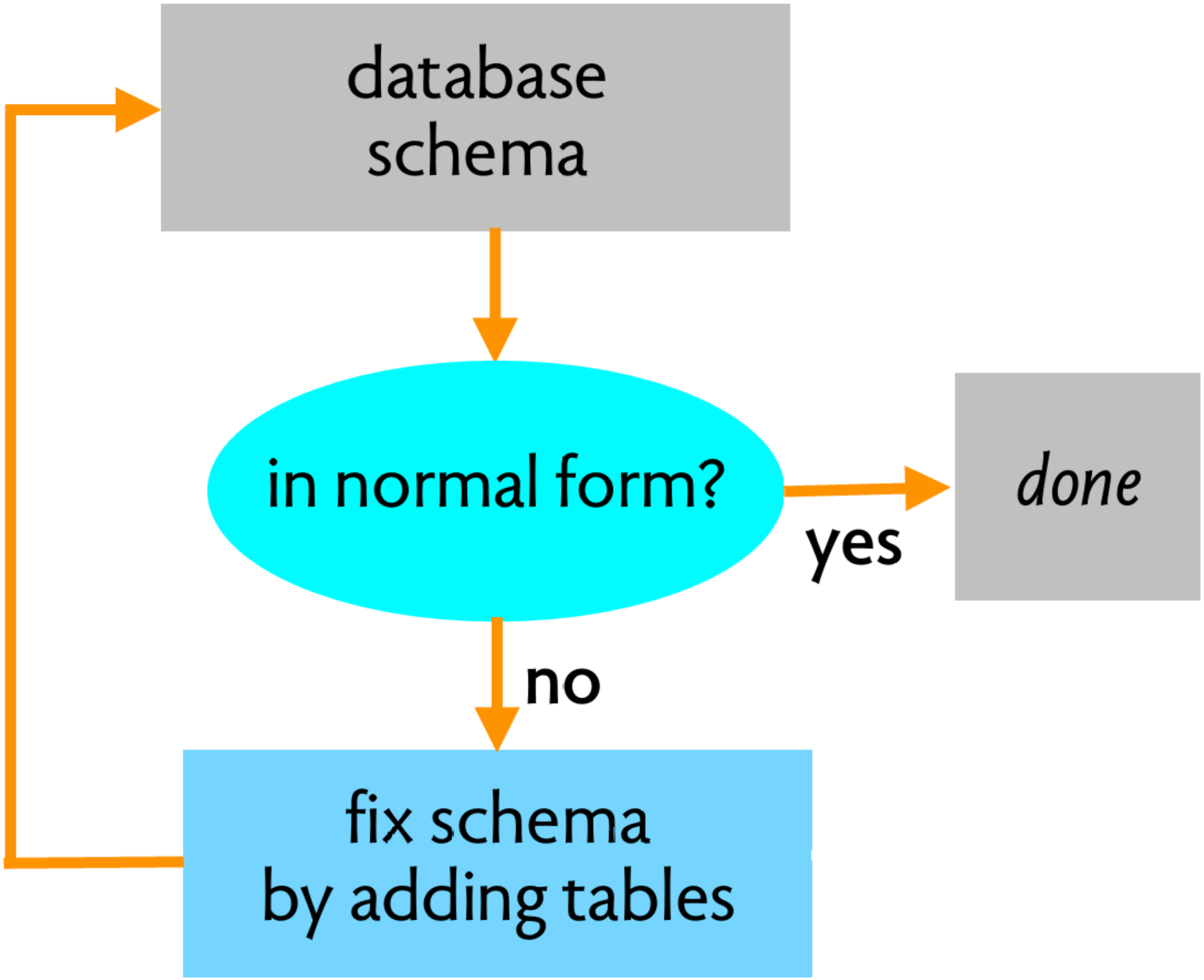
- don't actually want to store every object

example: web analytics

- may have set Visit
- but perhaps too many visits to save
- so instead store stats (eg, #visits)



traditional database design



database schema

in normal form?

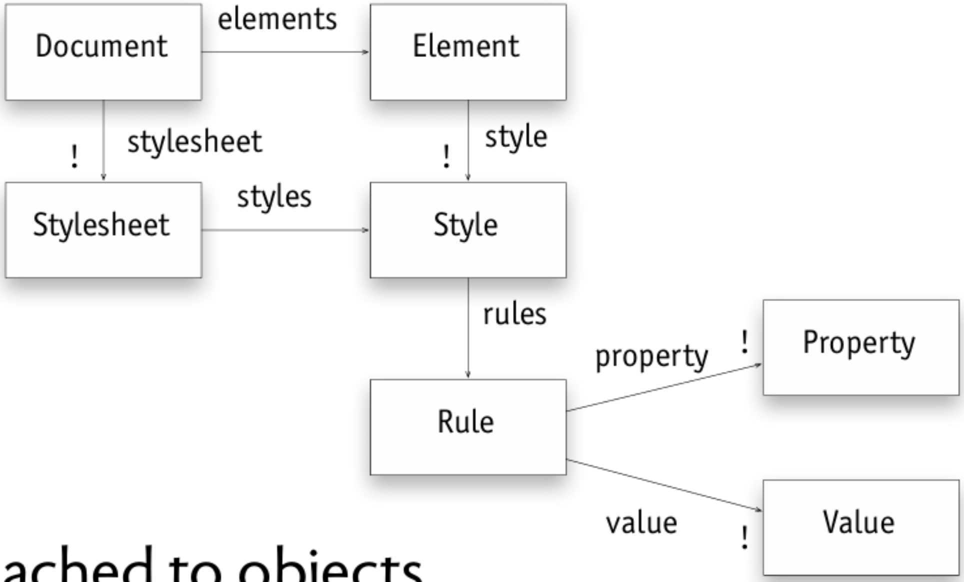
yes

done

no

fix schema by adding tables

idiom: stylesheet



problem

› maintain consistent styling

› make global changes easy

solution

› style: holds formatting rules, attached to objects

examples

› paragraph and character styles in Word, Indesign, Pages

› themes in Powerpoint

› cascading style sheets for HTML

refinements

› stylesheets: reuse and separation of sets of styles

› selectors: match objects implicitly

› inheritance: of properties from style to style

› overlapping styles: eg, character and paragraph styles

Chapter 8 – Software Testing

Topics covered

- ✧Development testing (T1-8.1, R2-8);
- ✧Test driven development (T1-8.2);
- ✧Release testing (T1-8.3);
- ✧ User testing (T1-8.4);

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

When you test software, you are trying to do two things:

1. To demonstrate to the developer and the customer that the **software meets its requirements** => validation testing
   - 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.
2. To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to **its specification** => defect testing
   - **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.

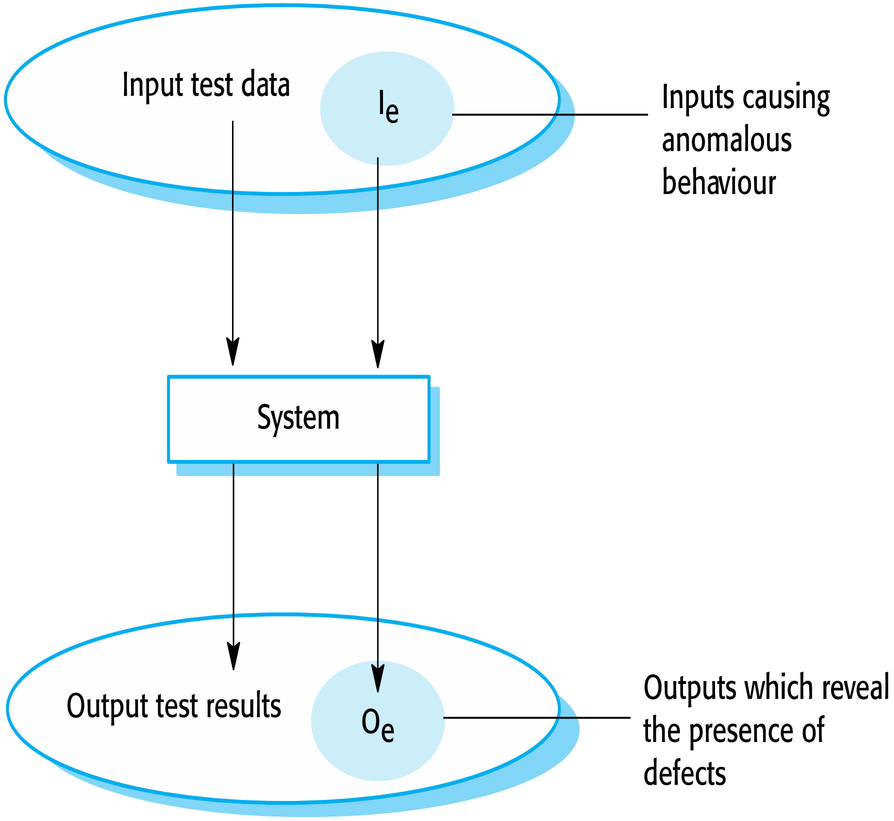
✧ **Defect testing**

- 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.

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 program meets its requirements

An input-output model of program testing


Input test data
Ie
Inputs causing anomalous behaviour
System
Output test results
Oe
Outputs which reveal the presence of defects

- Figure shows 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 Oe that are generated by the system in response to inputs in the set Ie.**
- **The priority in defect testing is to find those inputs in the set Ie because these reveal problems with the system.**
- **Validation testing involves testing with correct inputs that are outside Ie.** These stimulate the system to generate the expected correct outputs

Testing process goals

"Testing can only show the presence of errors, not their absence"

Testing is part of a broader process of software verification and validation (V & V).

Verification and validation are not the same thing, although they are often confused.

Verification vs validation

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

V & V confidence

- Aim of V & V is to establish confidence that the system is ‘fit for purpose’.
- Depends on **system’s purpose, user expectations and marketing environment**

1. **Software purpose**
    - The more critical the software, the more important it is that it is reliable.
    - For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a demonstrator system that prototypes new product idea
2. **User expectations**
    - Because of their previous experiences with buggy, unreliable software, users sometimes have low expectations of software quality.
    - They are not surprised when their software fails.
    - When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery.
    - However, as a software product becomes more established, users expect it to become more reliable. Consequently, more thorough testing of later versions of the system may be required
3. **Marketing environment**
    - When a software company brings a system to market, it must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system.
    - In a competitive environment, the company may decide to release a program before it has been fully tested and debugged because it wants to be the first into the market. If a software product or app is very cheap, users may be willing to tolerate a lower level of reliability

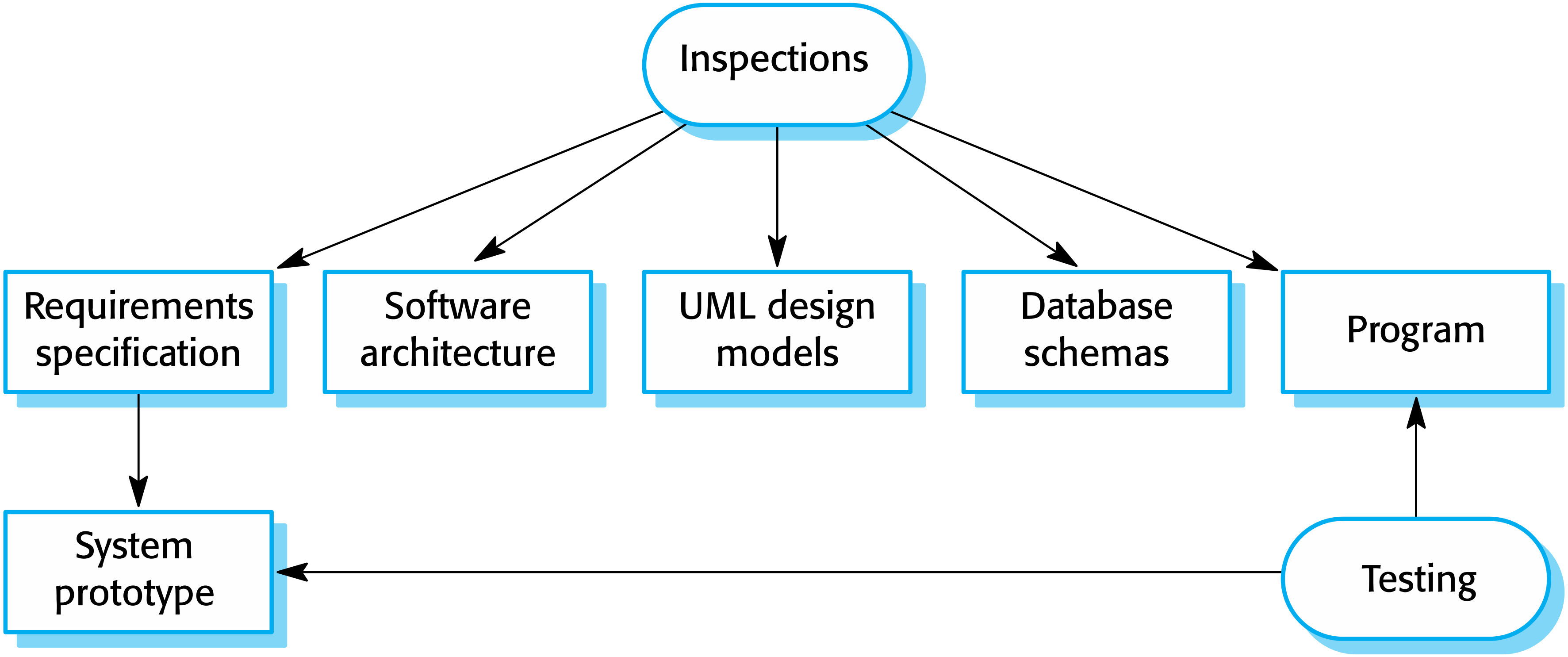
V & V confidence

- 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 “static” V & V techniques in which you don’t need to execute the software to verify it**

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.
- 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



Inspections

Requirements specification

Software architecture

UML design models

Database schemas

Program

System prototype

Testing

- Figure shows that **software inspections and testing support V & V at different stages** in the software process.
- The **arrows indicate the stages in the process where the techniques may be used**

Software inspections

- Inspections mostly **focus on the source code of a system**, but any readable representation of the software, such as its requirements or a design model, can be inspected.
- When you inspect a system, **you use knowledge of the system, its application domain, and the programming or modeling language** to discover errors

Advantages of inspections

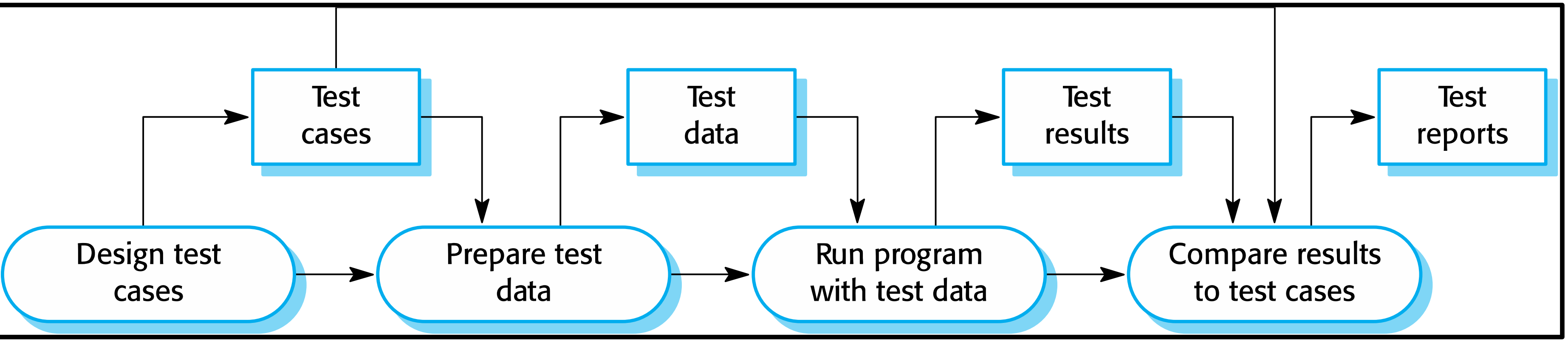
Software inspection has three advantages over testing:

1. During testing, errors can mask (hide) other errors.
   - When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side effects of the original error.
   - Because inspection doesn't involve executing the system, you don't have to worry about interactions between errors.
   - Consequently, a single inspection session can discover many errors in a system.
2. 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.
3. 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.
   - You can look for inefficiencies, inappropriate algorithms, and poor programming style that could make the system difficult to maintain and update.

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



- Figure above, is an abstract model of the traditional testing process, as used in plan driven development.
- **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 test 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.**

Stages of testing

1. **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.
2. **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 the system stakeholders.
3. **User testing**,
   - where users or potential users of a system test the system in their own environment.
   - For software products, the “user” may be an internal marketing group that decides if the software can be marketed, released and sold.
   - 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

T 8.1 Development testing

Development testing

- Development testing includes all testing activities that are carried out by the team developing the system.

1. **Unit testing**,
   - where individual program units or object classes are tested.
   - Unit testing should focus on testing the functionality of objects or methods.
2. **Component testing**,
   - where several individual units are integrated to create composite components.
   - Component testing should focus on testing component interfaces.
3. **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.

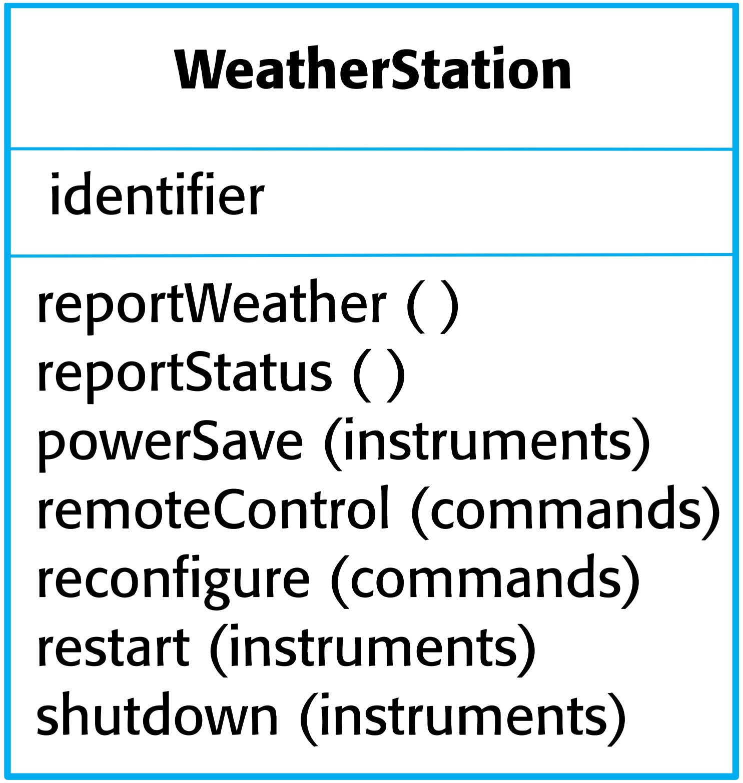
Unit testing

- ✧ Unit testing is the process of testing individual components in isolation.
  - Example, methods or object classes
- ✧ 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



WeatherStation

identifier

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

- It has a single attribute, which is its identifier.
- This is a constant that is set when the weather station is installed.
- You therefore only need a test that checks if it has been properly set up.
- You need to define test cases for all of the methods associated with the object such as **reportWeather** and **reportStatus.**
- Ideally, you should test methods in isolation, but, in some cases, test sequences are necessary.
- For example, to test the method that shuts down the weather station instruments (**shutdown**), you need to have executed the **restart** method.

Object class testing concern..

- Generalization or inheritance makes object class testing more complicated.
- You can’t simply test an operation in the class where it is defined and assume that it will work as expected in all of the subclasses that inherit the operation.
- The operation that is inherited may make assumptions about other operations and attributes.
- These assumptions may not be valid in some subclasses that inherit the operation.
- **You therefore have to test the inherited operation everywhere that it is used.**

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

1. **<u>A setup part:</u>**
   - **where you initialize the system with the test case, namely the inputs and expected outputs.**
2. **<u>A call part:</u>**
   - **where you call the object or method to be tested.**
3. **<u>An assertion part :</u>**
   - **where you compare the result of the call with the expected result.**
   - **If the assertion evaluates to true, the test has been successful if false, then it has failed.**

Choosing unit test cases

1. **The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.**
2. **If there are defects in the component, these should be revealed by test cases.**

✧ This leads to 2 types of unit test case:

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

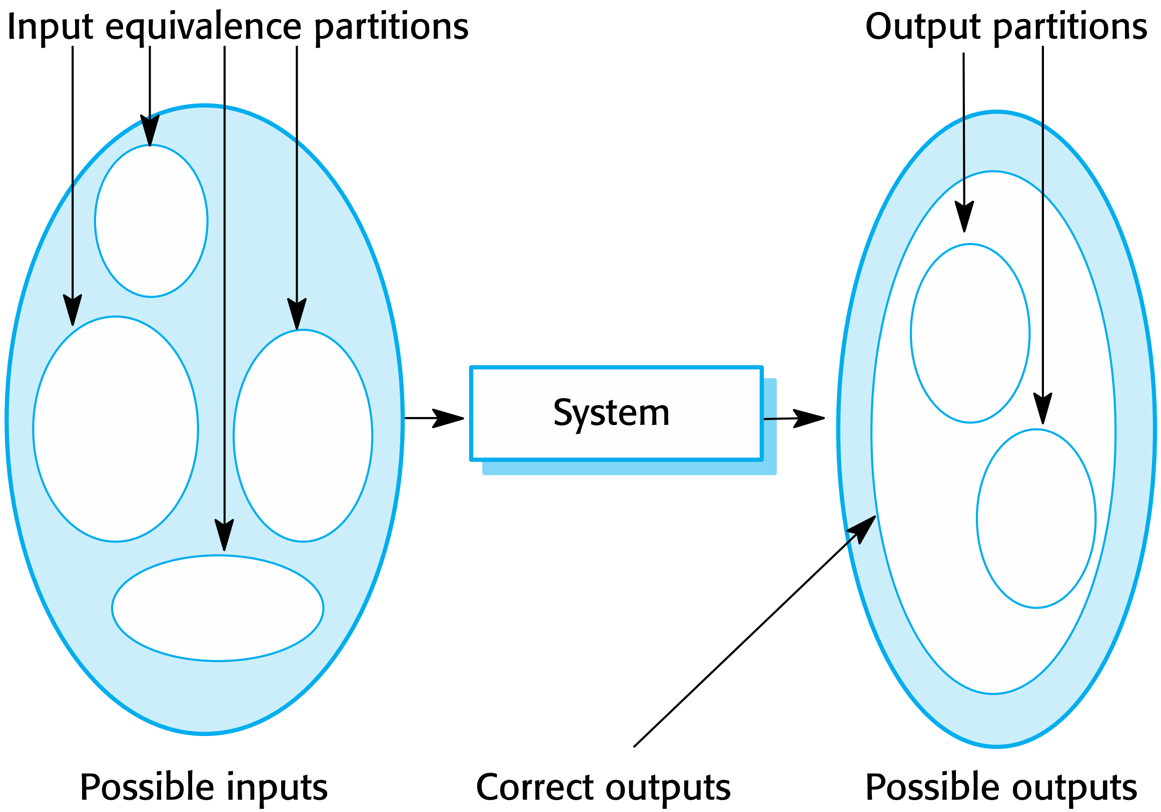
Testing strategies

1. **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.
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.

Equivalence partitioning



Equivalence partitioning

- 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, that is, responses to invalid inputs

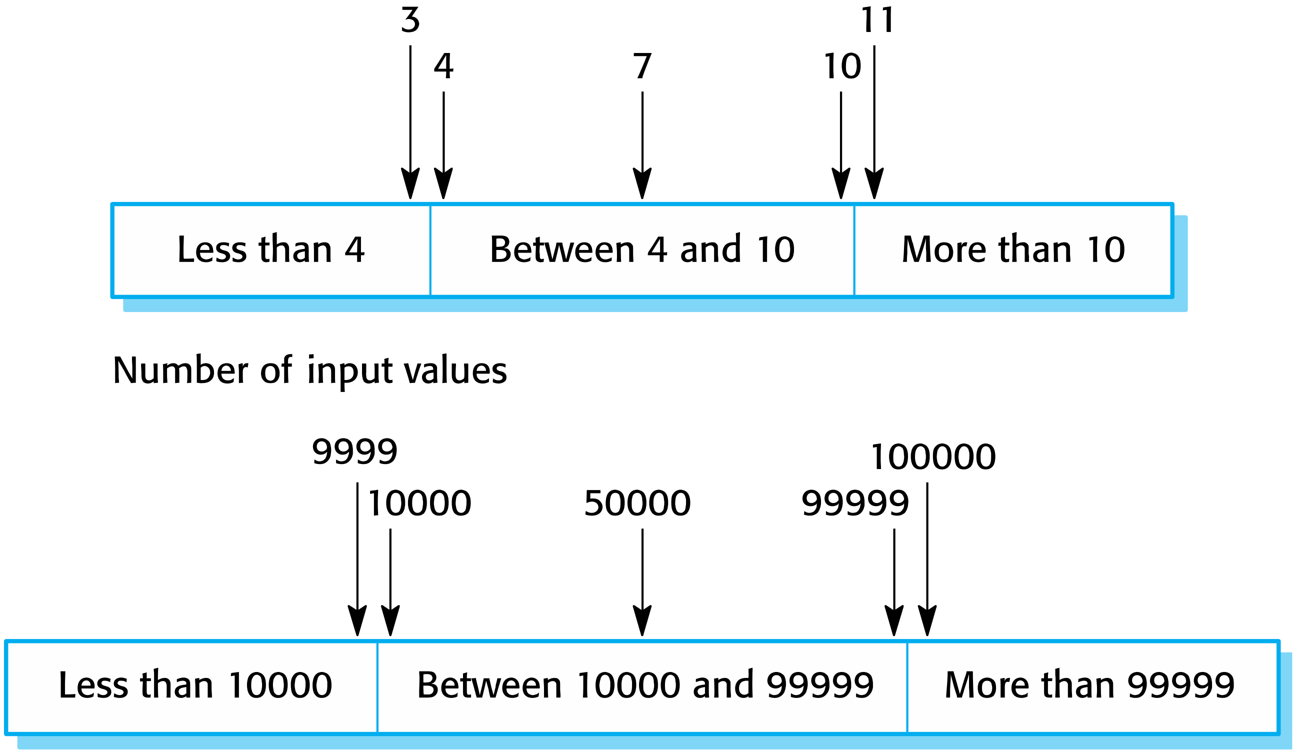
Equivalence partitions

- Once you have identified a set of partitions, you choose test cases from each of these partitions.
- **A good rule of thumb for test-case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition**.
- 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) and so are sometimes overlooked by developers.
- Program failures often occur when processing these atypical values

Equivalence partitions

- 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 four to eight 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



Number of input values

Input values

Equivalence partitions

✧ Example

1. Age group between 9 to 60
2. Test score :S(90-99), A (80-89), B(70-79), C(60-69), D(50-59), E(40-49), F(<40)
3. Password length between 8 to 16
4. Bank Withdrawal amount between 10000 to 100000 Rs

Equivalence partitions

- Equivalence partitioning is an effective approach to testing because it helps account for errors that programmers often make when processing inputs at the edges of partitions.
- You can also use testing guidelines to help choose test cases.
- Guidelines encapsulate knowledge of what kinds of test cases are effective for discovering errors.
- For example, when you are testing programs with sequences, arrays, or lists, guidelines that could help reveal defects include: (@next slide)

Testing guidelines (sequences)

1. **Test software with sequences that have only a single value.**
   - Programmers naturally think of sequences as made up of several values, and sometimes they embed this assumption in their programs.
   - Consequently, if presented with a single-value sequence, a program may not work properly.

2. **Use different sequences of different sizes in different tests**.

   - This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.

3. **Derive tests so that the first, middle, and last elements of the sequence are accessed.**

   - This approach reveals problems at partition boundaries.

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.

Component testing: Interface testing

- ✧ Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- ✧ Interface types

1. **Parameter interfaces**

   ○ These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.Shared memory interfaces Block of memory is shared between procedures or functions.

2. **Shared memory interfaces**

○ These are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem and retrieved from there by other subsystems. This type of interface is used in embedded systems, where sensors create data that is retrieved and processed by other system components.

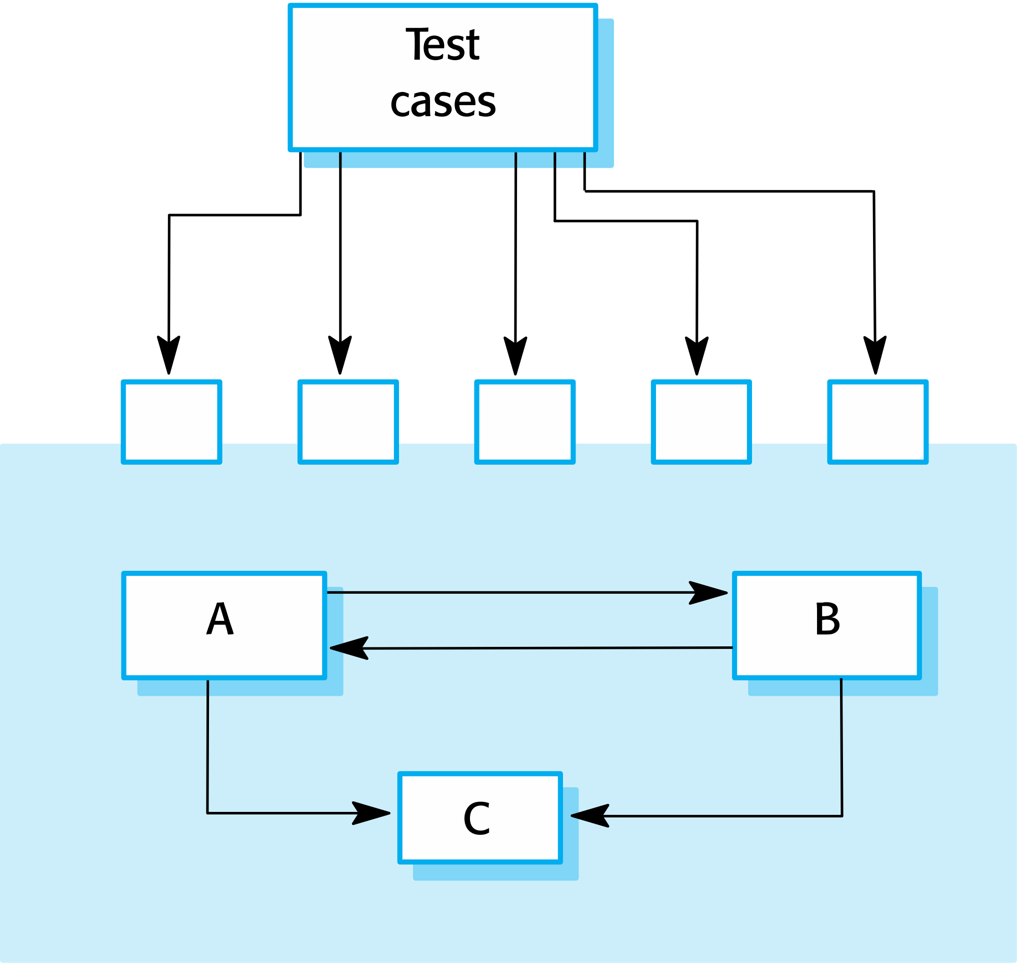
3. **Procedural interfaces**

○ These are interfaces in which one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.

4. **Message passing interfaces**

○ These are interfaces in which one component requests a service from another component 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

Component testing: Interface testing



- 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.

Component 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.
  - This type of error is common in parameter interfaces, where parameters may be of the wrong type or be passed in the wrong order, or the wrong number of parameters may be passed.
- ✧ Interface misunderstanding
  - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
  - The called component does not behave as expected, which then causes unexpected behavior in the calling component. For example, a binary search method may be called with a parameter that is an unordered array. The search would then fail
- ✧ Timing errors
  - The called and the calling component operate at different speeds and out-of-date information is accessed.
  - 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 because the producer of the information has not updated the shared interface information.

Component testing: Interface testing guidelines

1. Examine the code to be tested and identify each call to an external component. Design a set of tests in which the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.
2. Where pointers are passed across an interface, always test the interface with null pointer parameters.
3. Where a component is called through a procedural interface, design tests that deliberately cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.
4. 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.
5. Where several components interact through shared memory, design tests that vary the order in which these components are activated. These tests may reveal implicit assumptions made by the prog

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.

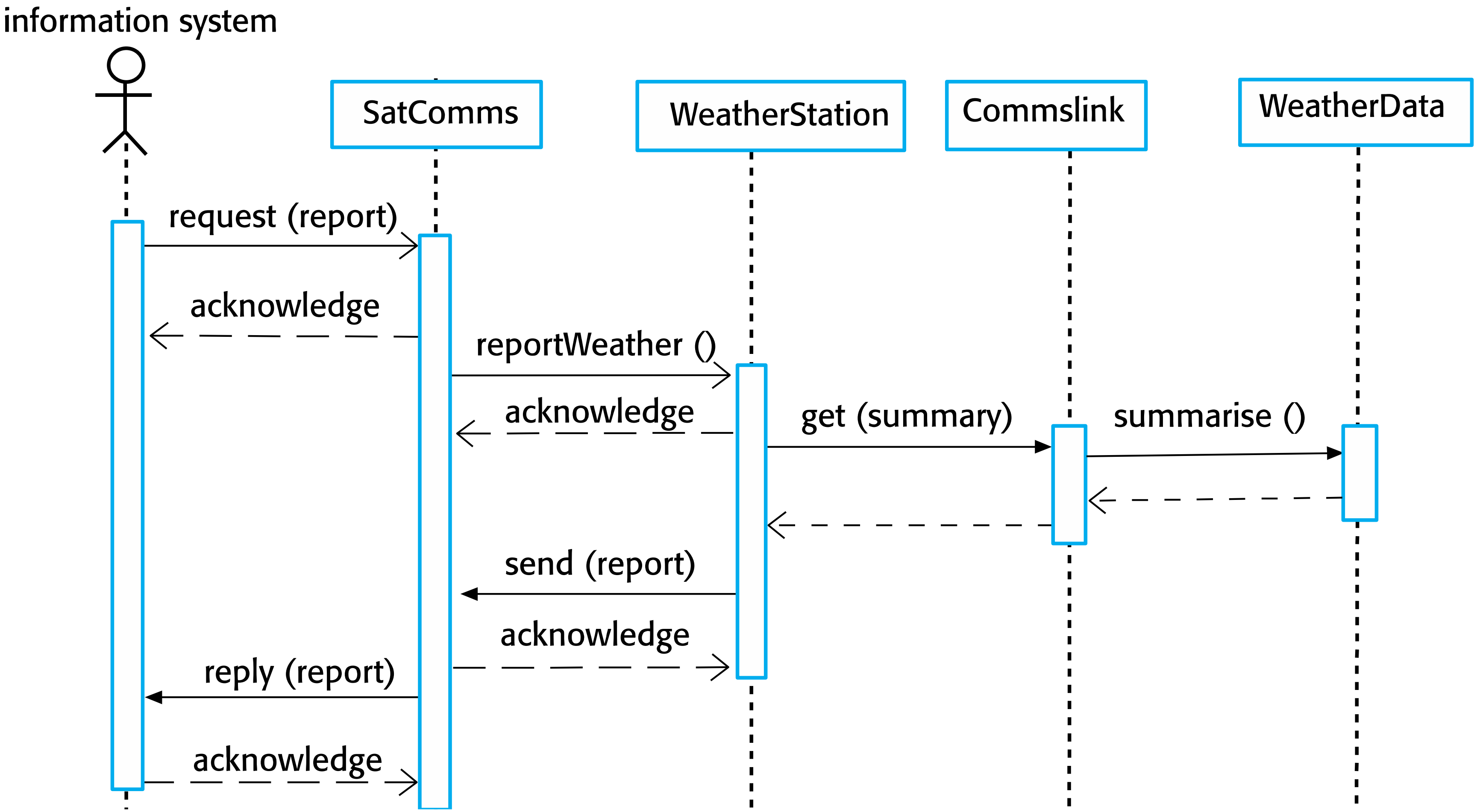
System and component testing

1. 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.
2. 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

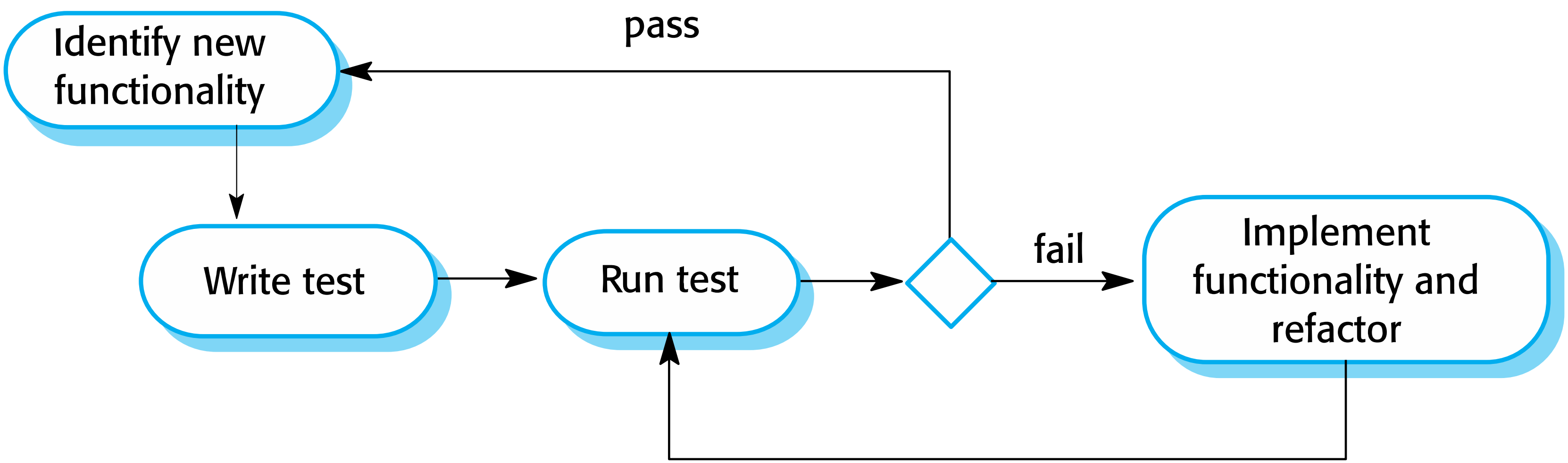
- ✧ Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- ✧ Examples of testing policies:
  - ▪ All system functions that are accessed through menus should be tested.
  - ▪ Combinations of functions (e.g. text formatting) that are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


Identify new functionality
pass
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

1. 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.

2. 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.

3. 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.

4. Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.

5. 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.

A usage scenario for the Mentcare system

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

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

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

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

Features tested by scenario

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

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

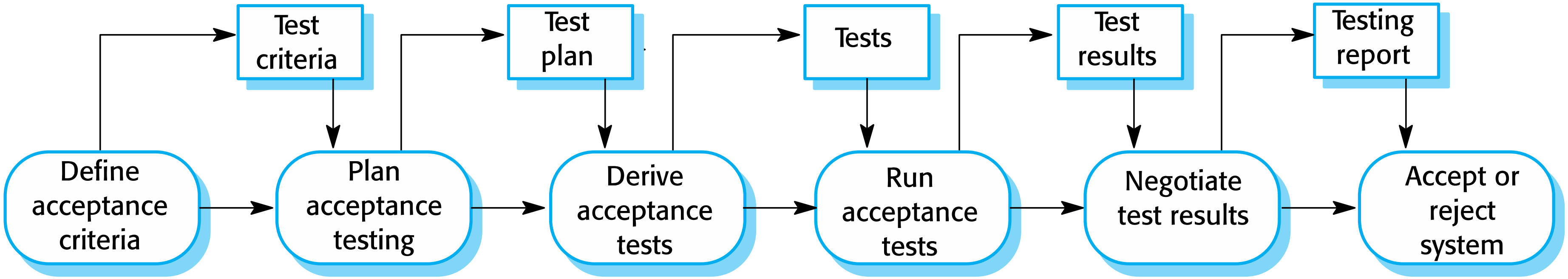
User testing

- ✧ 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.