Implementation and Testing (9 hours)

Module III

Object-oriented design using the UML

- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- 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.
- Design and implementation are closely linked, and you should normally take implementation issues into account when developing a design.

Two aims:

- 1. To show how system modeling and architectural are put into practice in developing an object-oriented software design.
- 2. To introduce important implementation issues that are not usually covered in programming books.

These include software reuse, configuration management and open-source development.

Object-oriented design using the UML

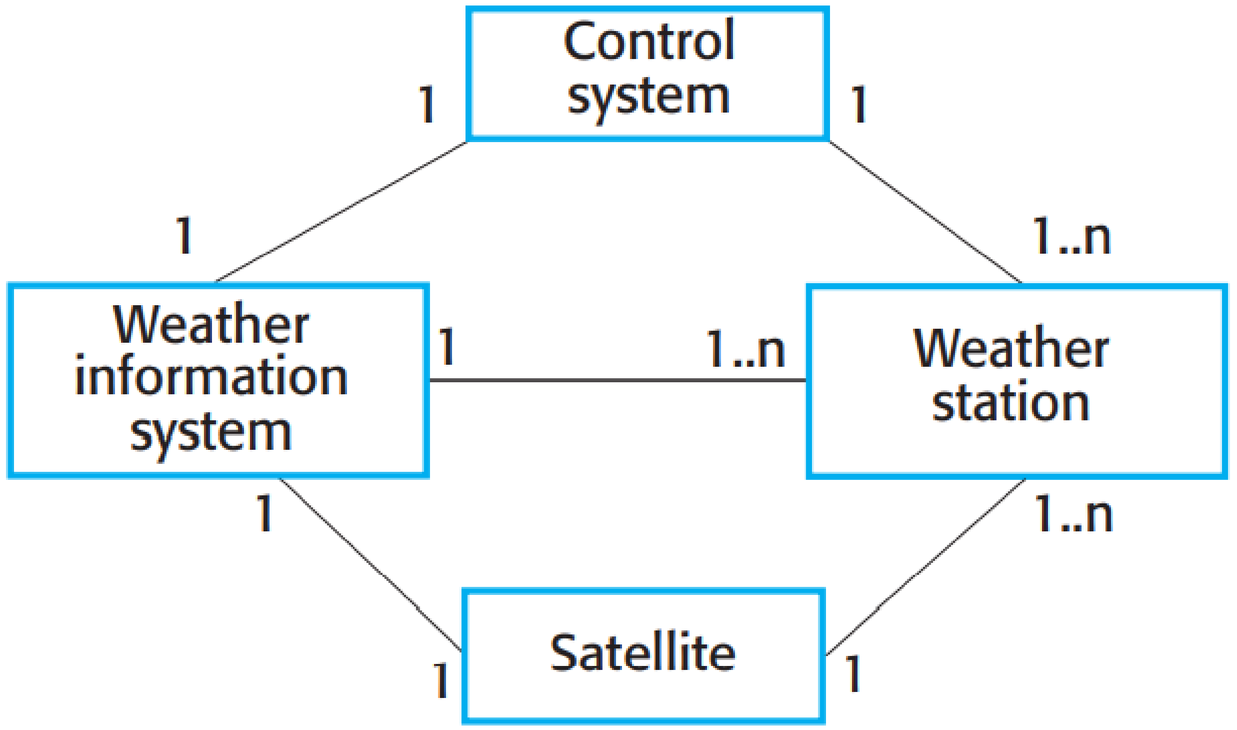
- An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state
- Object-oriented design processes involve designing object classes and the relationships between these classes.
- Objects include both data and operations to manipulate that data.

System context and interactions

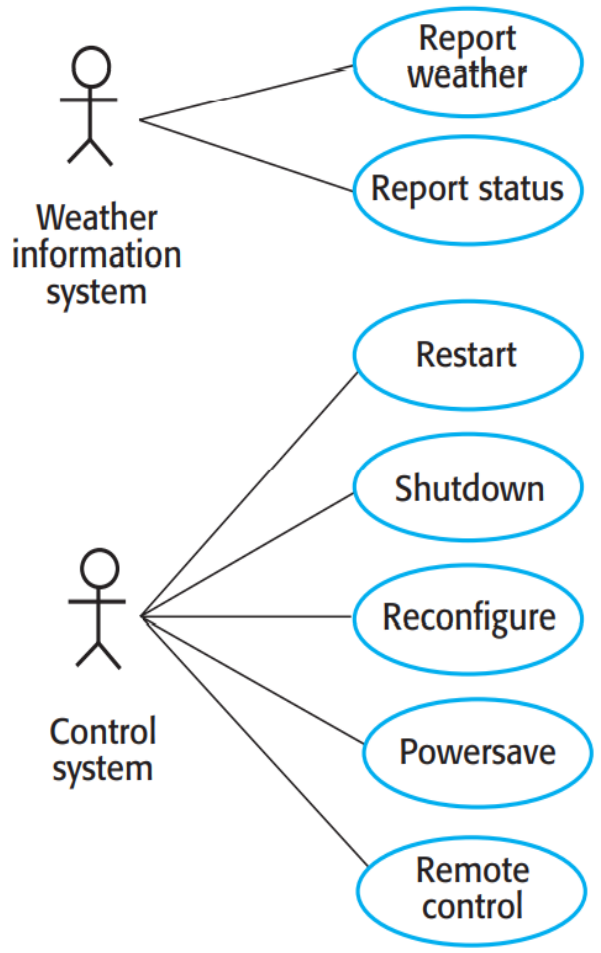
- System context models and interaction models present complementary views of the relationships between a system and its environment:
- A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

- The context model of a system may be represented using associations.
- Associations simply show that there are some relationships between the entities involved in the association.
- You can document the environment of the system using a simple block diagram, showing the entities in the system and their associations.

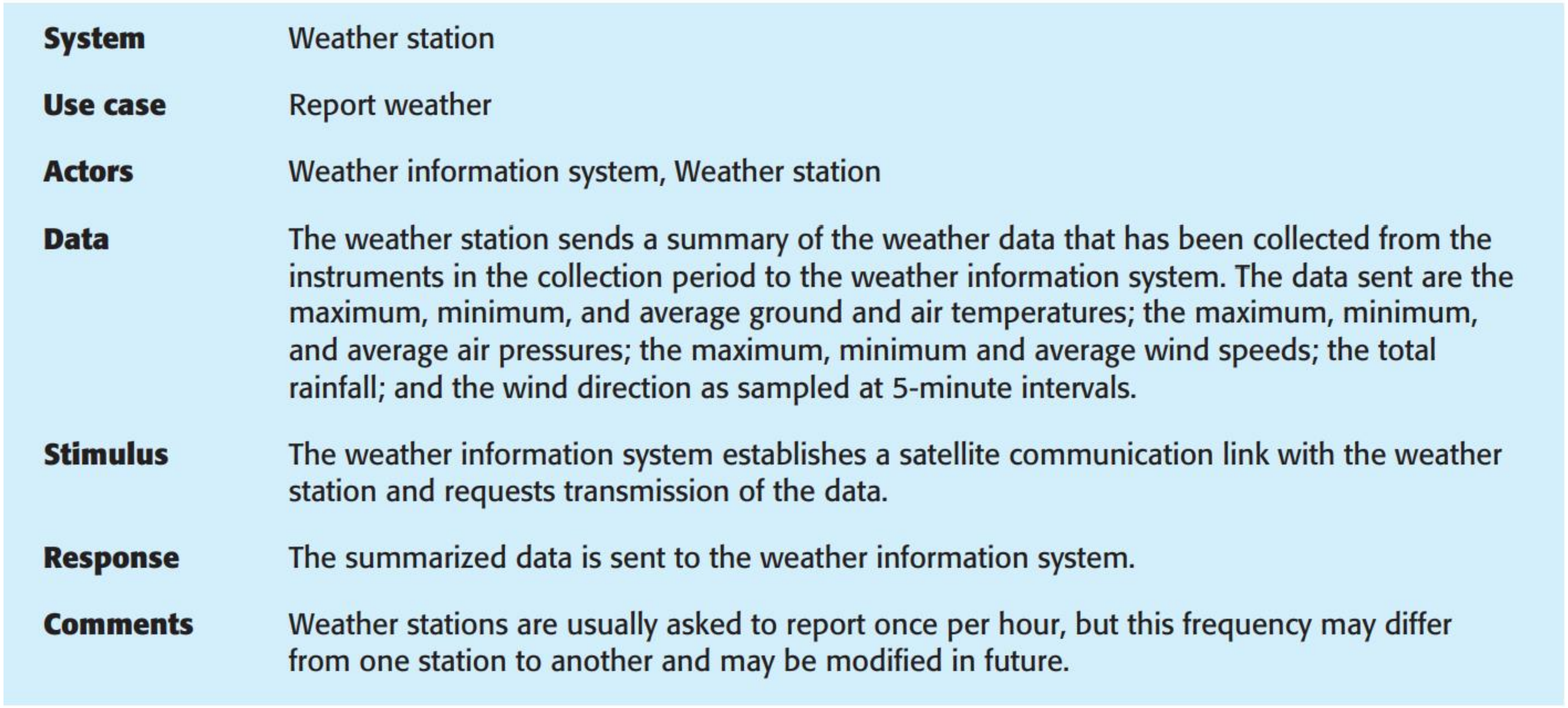
System context for the weather station



Weather station use cases



Usecase description-Report weather



System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	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 5-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 future.

Architectural design

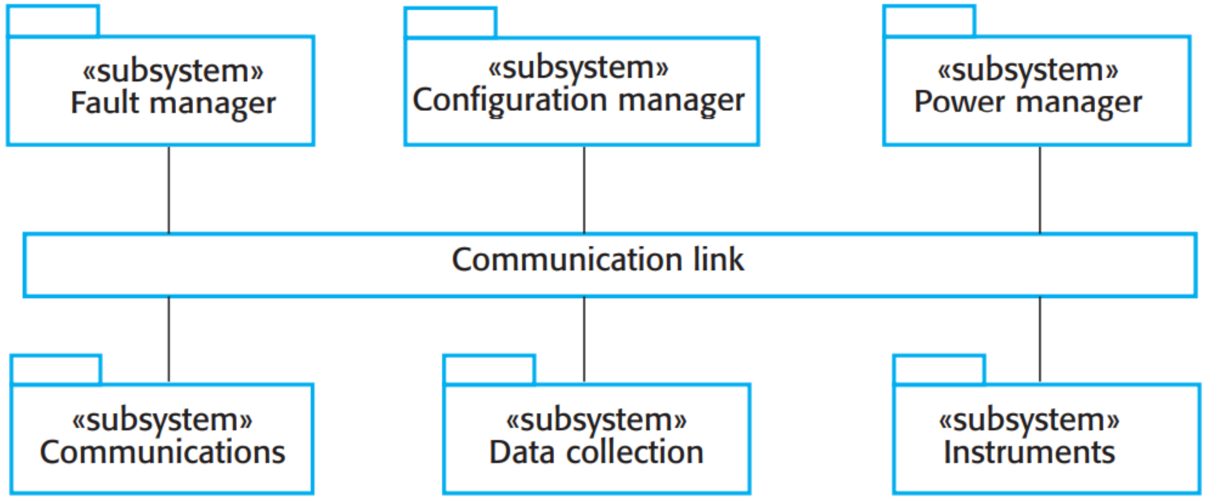
- Once the interactions between the software system and the system's environment have been defined,
- Identify the major components that make up the system and their interactions. Then design the system organization using an architectural pattern such as a layered or client–server model

High-level architecture of weather station

The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure(Communication link here)

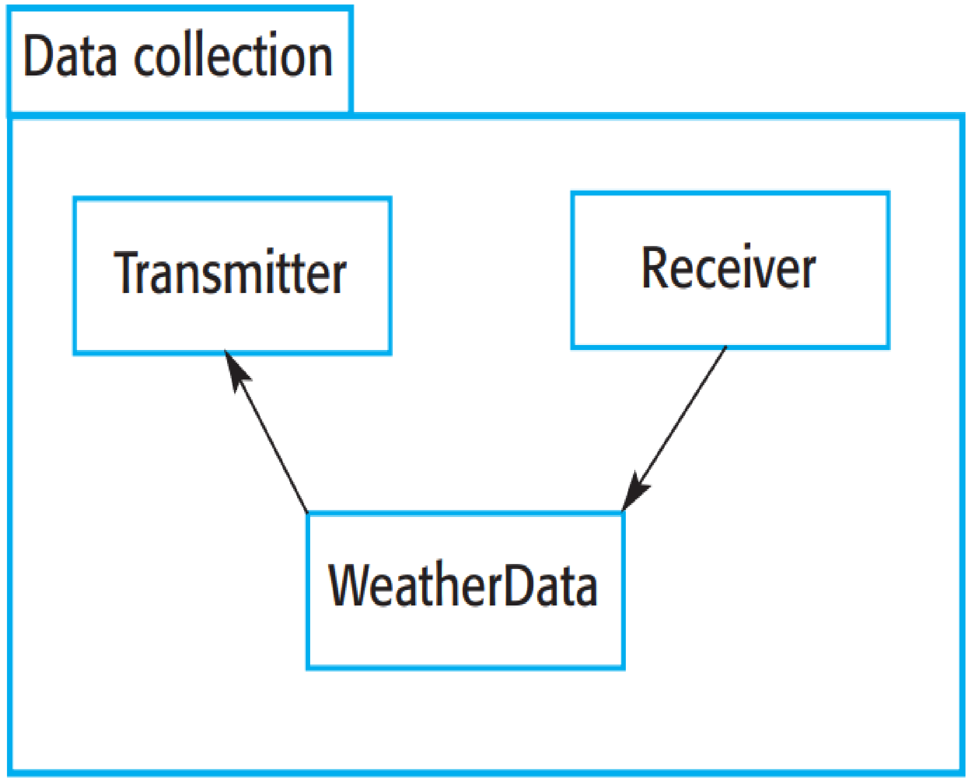
Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them. This "listener model" is a commonly used architectural style for distributed systems.

- **Benefit** :it is easy to support different configurations of subsystems because the sender of a message does not need to address the message to a particular subsystem


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

Architecture of data collection system

- The Transmitter and Receiver objects are concerned with managing communications
- The WeatherData 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

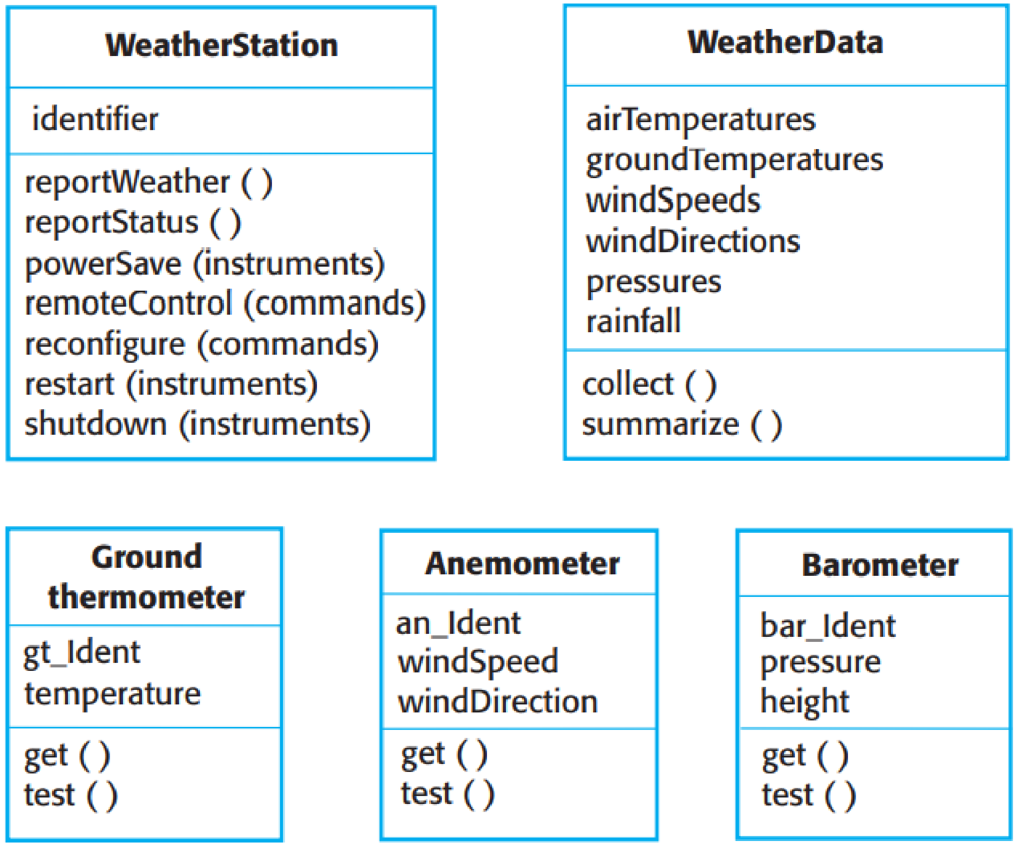
Various ways of identifying object classes in object-oriented systems :

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 etc.

3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn.

Weather station objects



Design models

- Two kinds of design model:

  1. Structural models, which describe the static structure of the system using object classes and their relationships.

  2. Dynamic models, which describe the dynamic structure of the system and show the expected runtime interactions between the system objects.

Three UML model types:

1. Subsystem models, which show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are structural models.

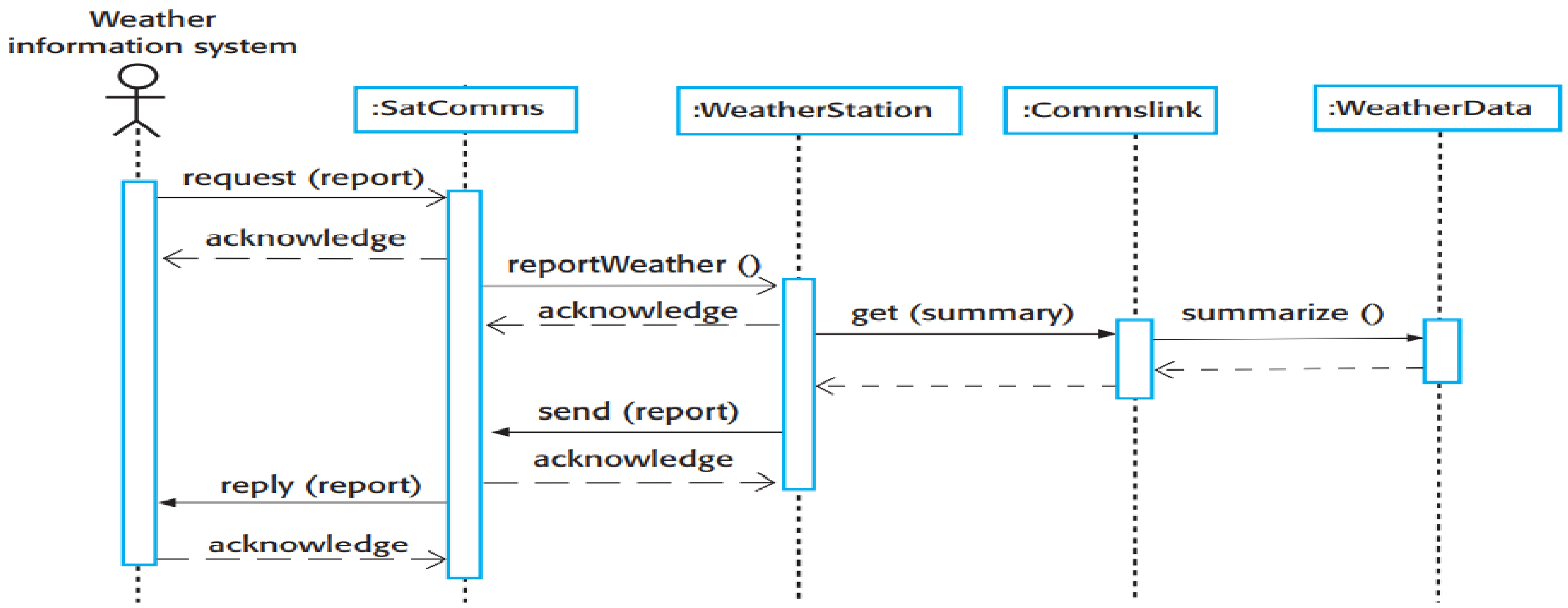
2. Sequence models, which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.

3. State machine models, which show how objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.

- A subsystem model is a useful static model that shows how a design is organized into logically related groups of objects.

- Sequence models are dynamic models that describe, for each mode of interaction, the sequence of object interactions that take place.

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.

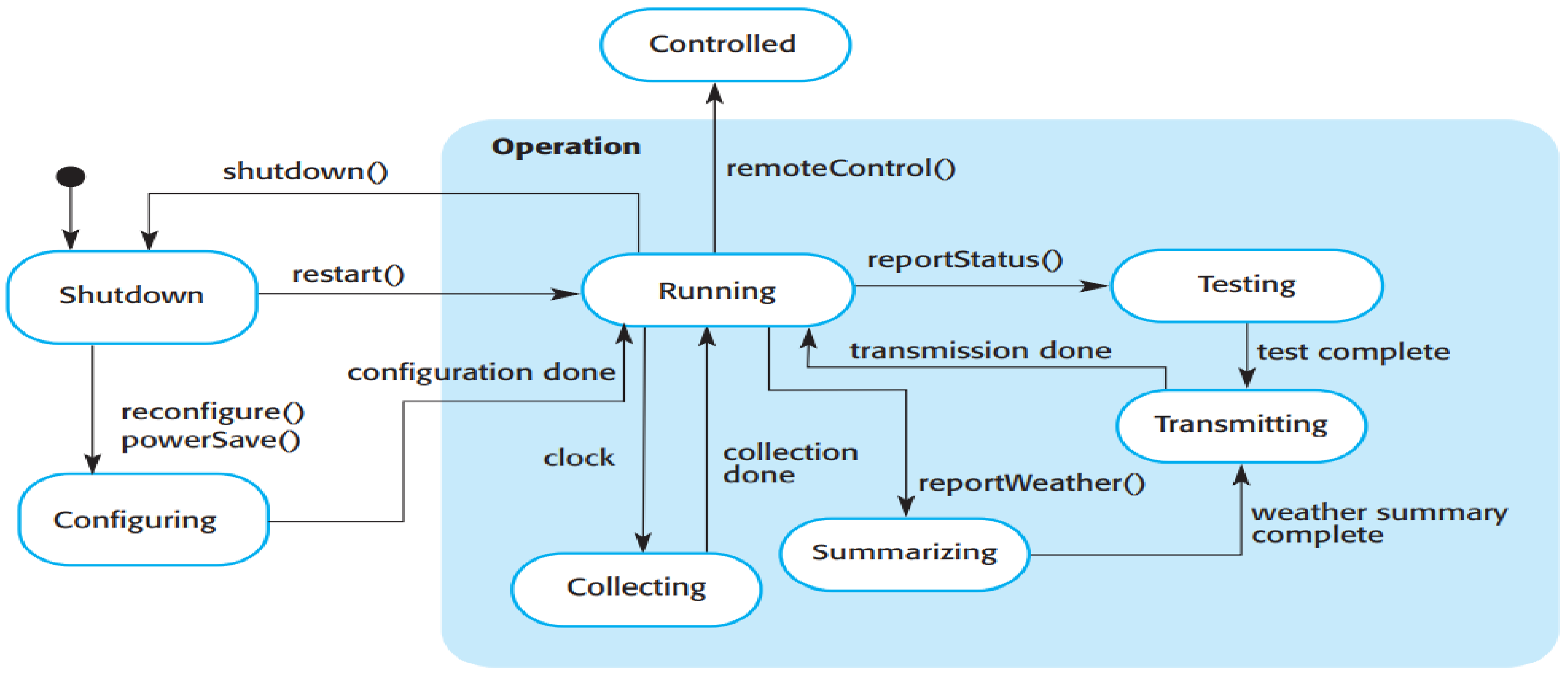
3. **WeatherStation** sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.

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

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.

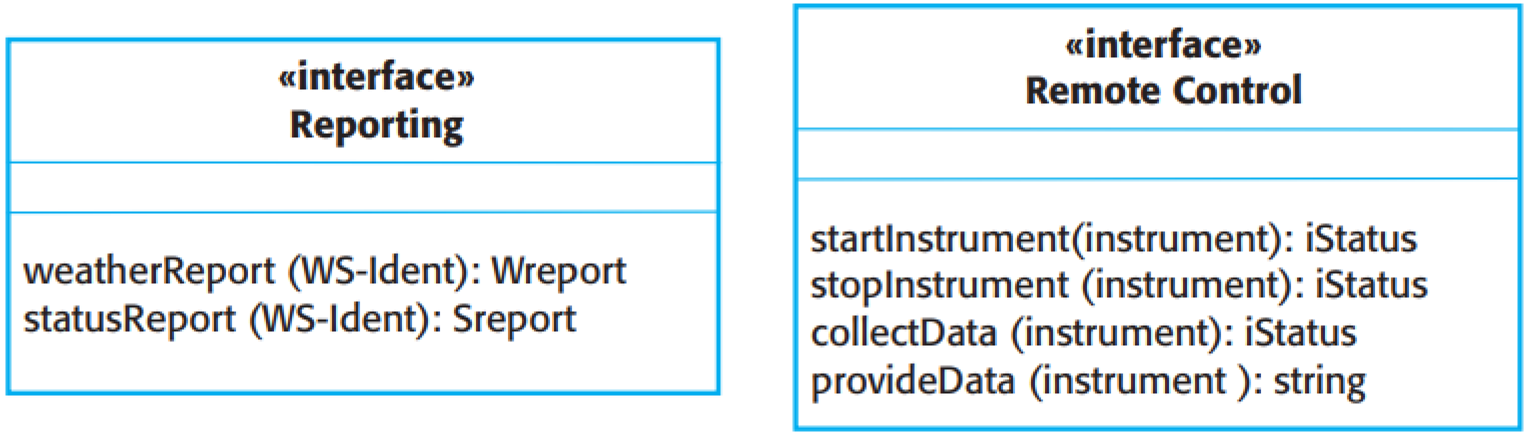
3. If a reportWeather() message is received, the system moves to the Summarizing state. When the summary is complete, the system moves to a Transmitting state where the information is transmitted to the remote system. It then returns to the Running state.

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

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

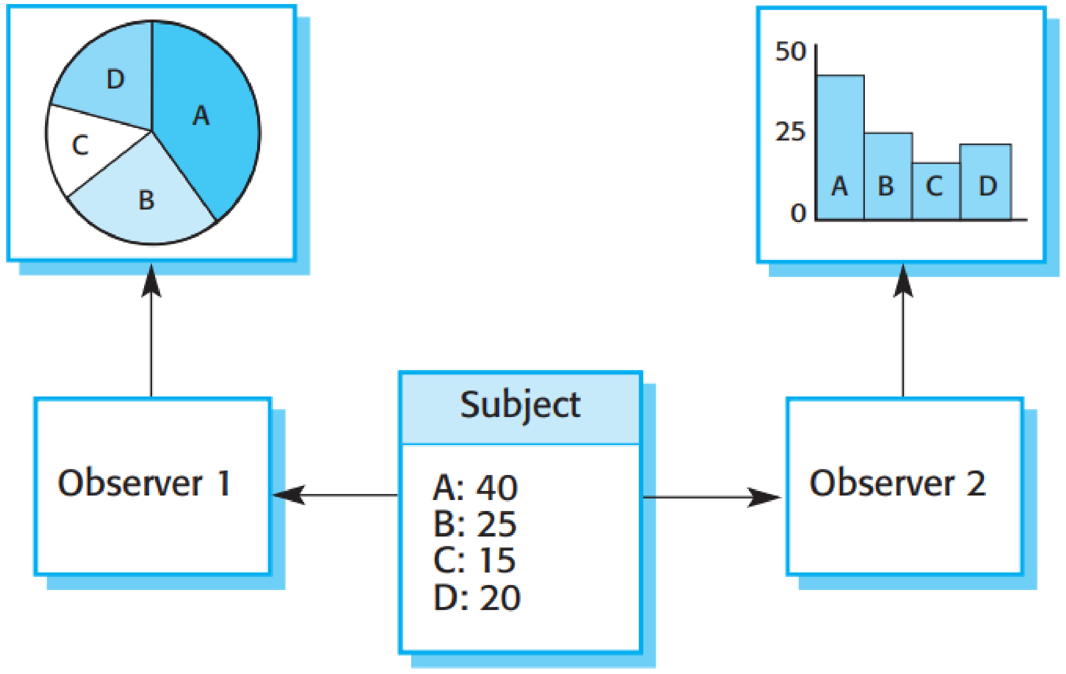
- The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. The pattern is not a detailed specification.
- Patterns have made a huge impact on object-oriented software design
- They have become a vocabulary for talking about a design
- Patterns are a way of reusing the knowledge and experience of other designers. Design patterns are usually associated with object-oriented design.
- The general principle of encapsulating experience in a pattern is one that is equally applicable to any kind of software design.

Four essential elements of design patterns

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships and their responsibilities.It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
4. A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

Two different graphical presentations of the same dataset:

- Graphical representations are normally used to illustrate the object classes in patterns and their relationships.
- These supplement the pattern description and add detail to the solution description.



IMPLEMENTATION ISSUES

- Software engineering includes all of the activities involved in software development from the initial requirements of the system through to maintenance and management of the deployed system.
- A critical stage of this process is, of course, system implementation, where you create an executable version of the software.
- Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization.

Aspects of implementation

1. 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.
2. Configuration management:
   - During the development process, many different versions of each software component are created. If you don’t keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.
3. 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)

1.Reuse

Different Levels of reuse:

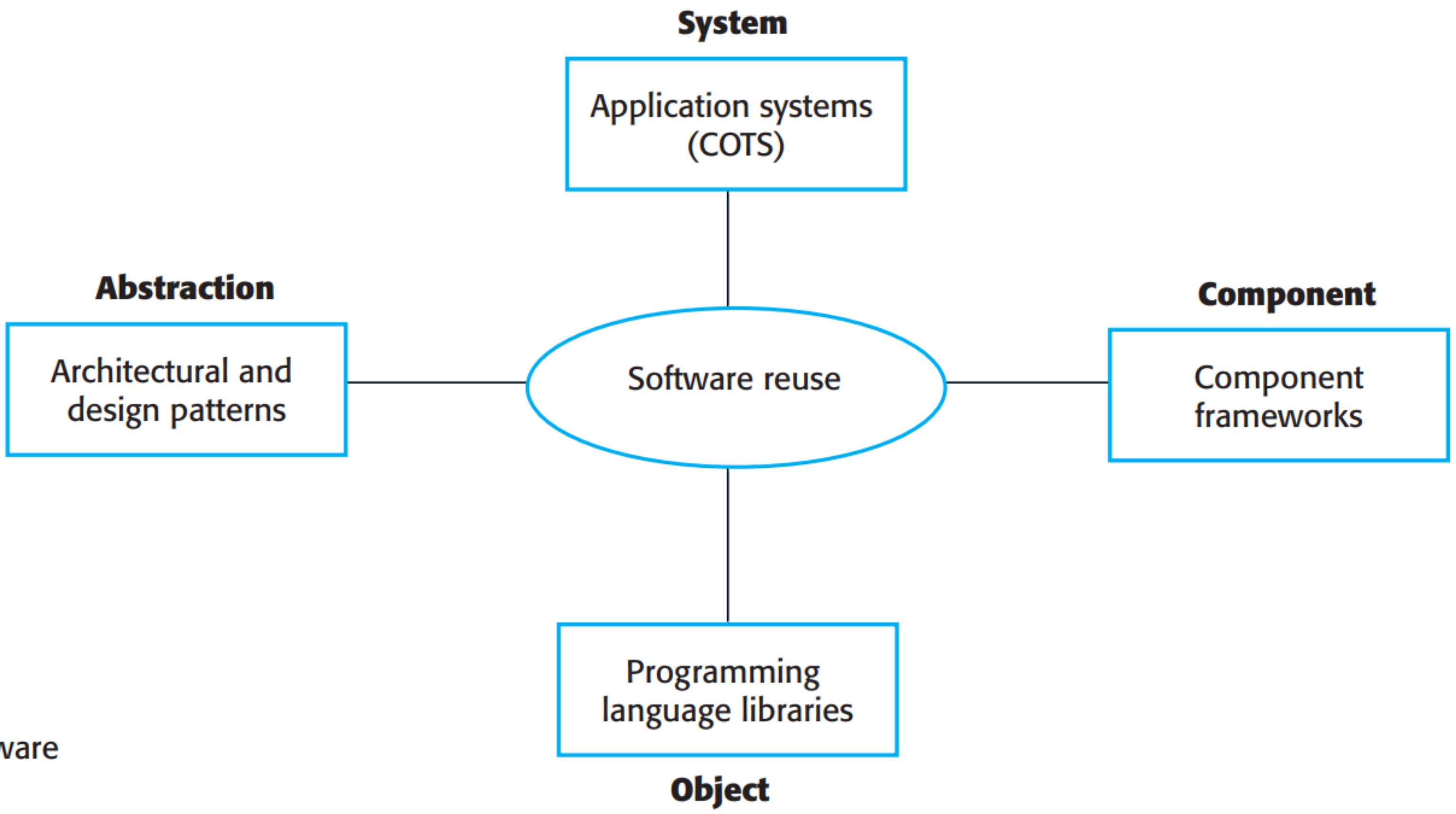
1**. The abstraction level**: At this level, you don’t reuse software directly but rather use knowledge of successful abstractions in the design of your software.eg. Design patterns and architectural patterns

2. **The object level:** At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need. For example, JavaMail library.

3. **The component level:** Components are collections of objects and object classes that operate together to provide related functions and services. An example of component-level reuse is where you build your user interface using a framework.

4. **The system level:** At this level, you reuse entire application systems. This function usually involves some kind of configuration of these systems. This may be done by adding and modifying code or by using the system’s own configuration interface

Software reuse


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

oftware

Costs associated with reuse:

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-theshelf 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.

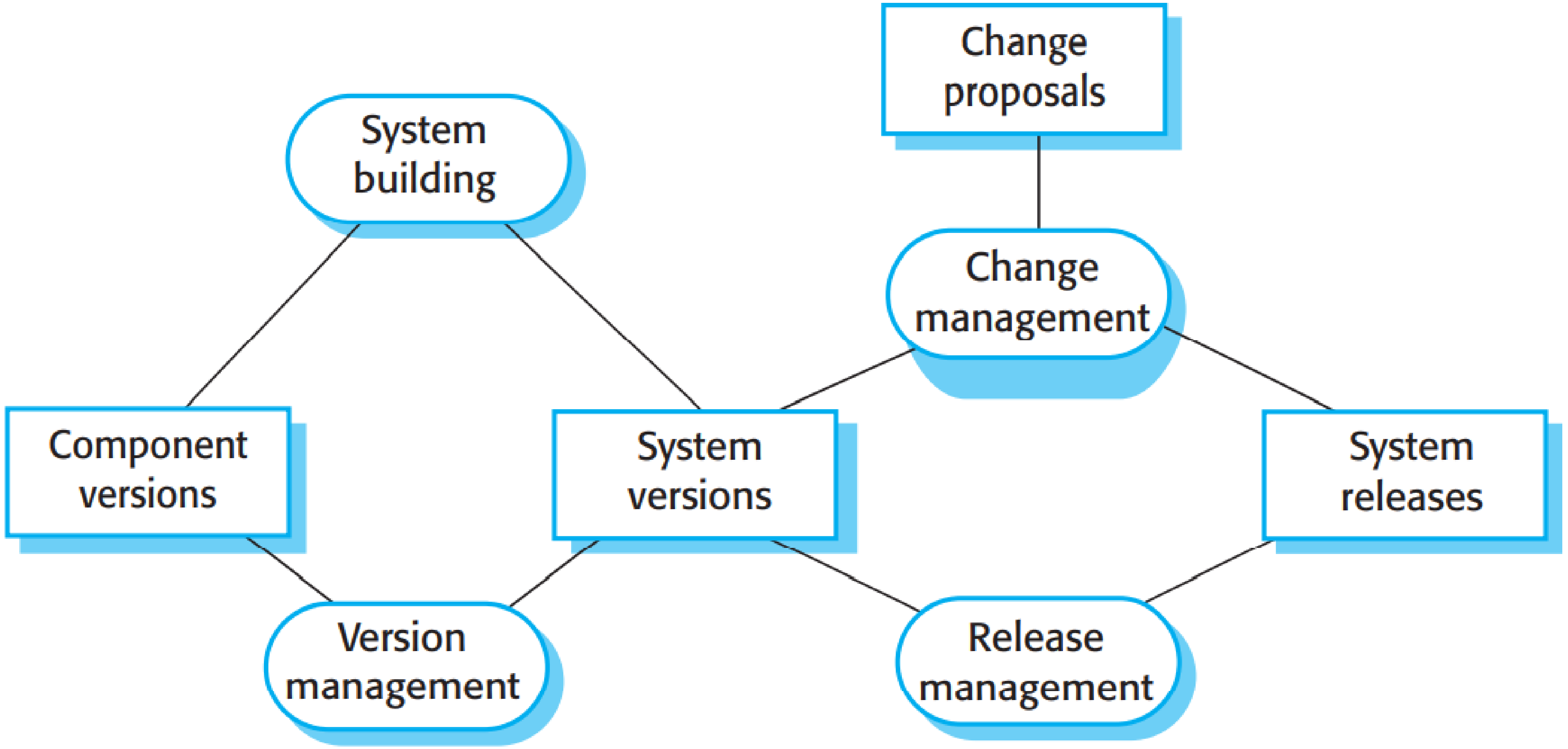
- By reusing existing software, you can develop new systems more quickly, with fewer development risks and at lower cost. As the reused software has been tested in other applications, it should be more reliable than new software
- How to reuse existing knowledge and software should be the first thing you should think about when starting a software development project.

2.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.
- Software configuration management tools support each of the above activities. These tools are usually installed in an integrated development environment, such as Eclipse.

Four fundamental 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. This description is then used to build a system automatically by compiling and linking 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.


Change proposals
System building
Change management
Component versions
System versions
System releases
Version management
Release management

3. Host-target development

- Most professional software development is based on a host-target model .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 (host) and an execution platform (target).
- 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.
- Simulators are often used when developing embedded systems.
- Simulators speed up the development process for embedded systems

A software development platform should provide a range of tools to support software 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.

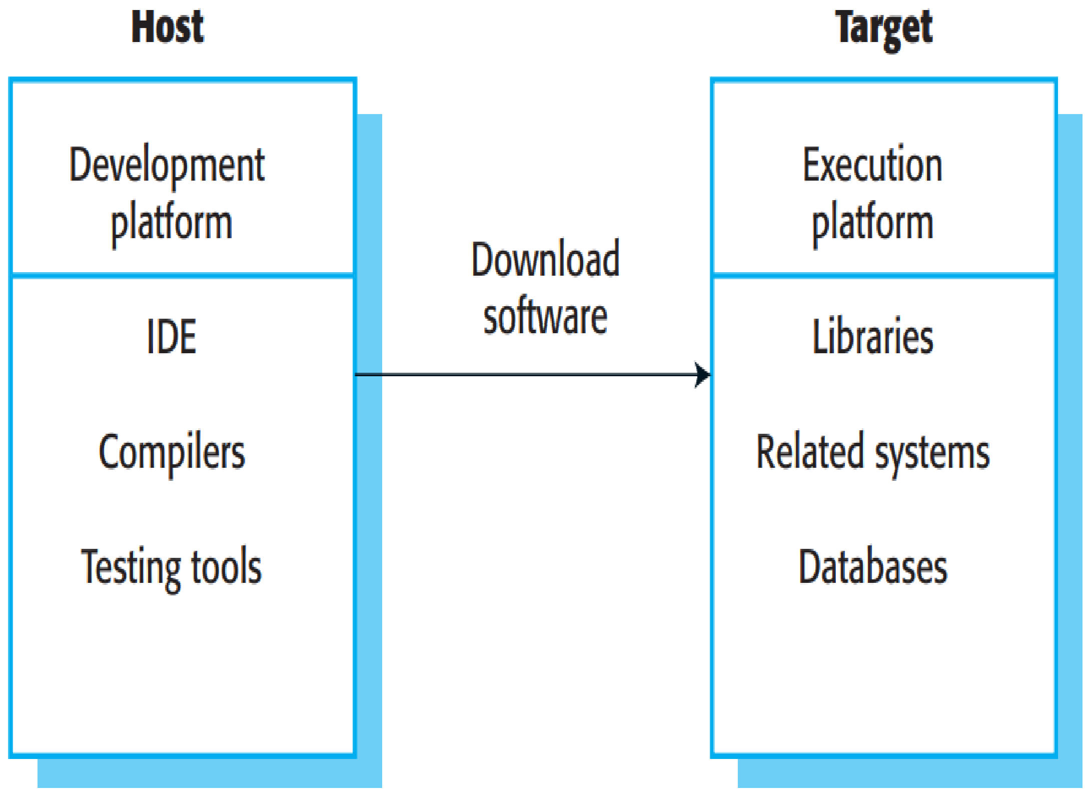
5. Tools to support refactoring and program visualization.

6. Configuration management tools to manage source code versions and to integrate and build systems.

Open source development

A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed and integration mechanisms that allow tools to work together.

The best-known general-purpose IDE is the Eclipse environment (http://www.eclipse.org).



Issues

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.

Open-source licensing

- A fundamental principle of open-source development is that source code should be freely available.
- Legally, the developer of the code owns the code. They can place restrictions on how it is used by including legally binding conditions in an open-source software license
- Licensing issues are important because if you use open-source software as part of a software product, then you may be obliged by the terms of the license to make your own product open source..
- The open-source approach is one of several business models for software.

Open source development

- Linux operating system
- widely used as a server system and as a desktop environment .
- Java, Apache Web server, my Sql database management system
- Benefits
- Fairly cheap or free to acquire open source software.
- Mature open source systems are usually very reliable.
- Bugs are discovered and repaired more quickly than is usually possible with proprietary software.

Most open-source licenses are variants of one of three general models:

1. The GNU General Public License (GPL).

This is a so-called reciprocal license that simplistically 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).

This 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. However, if you change the licensed component, then you must publish this as open source.

3. The Berkley Standard Distribution (BSD) License.

This is a nonreciprocal 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. If you use open-source components, you must acknowledge the original creator of the code. eg.The MIT license

Companies managing projects that use open source should:

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 developers 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.