Software Engineering

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

Process stages

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

System context and interactions

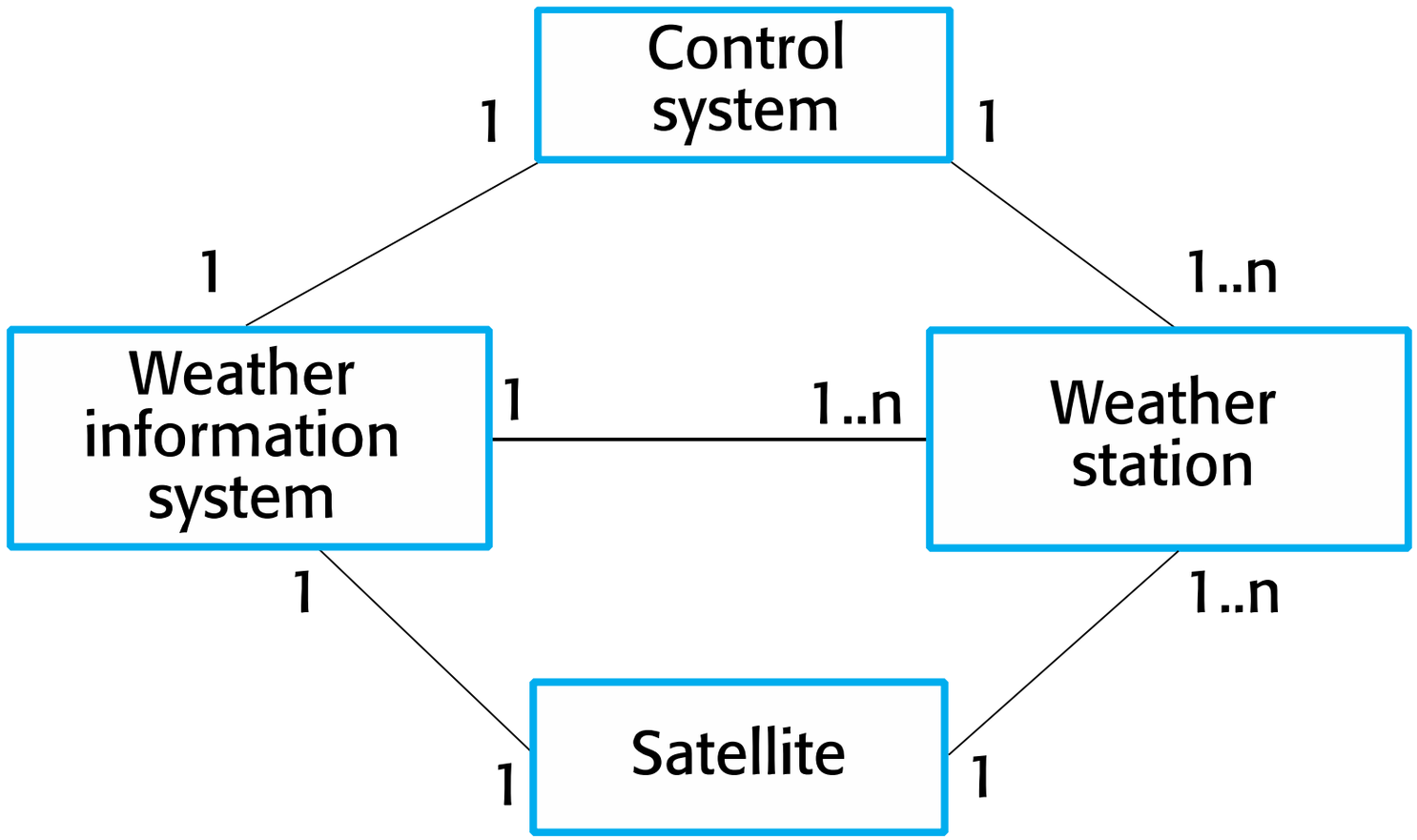
- ***Understanding the relationships between the software that is being designed and its external environment*** is essential,
  - for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- ***Understanding of the context*** also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

Context and interaction models

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

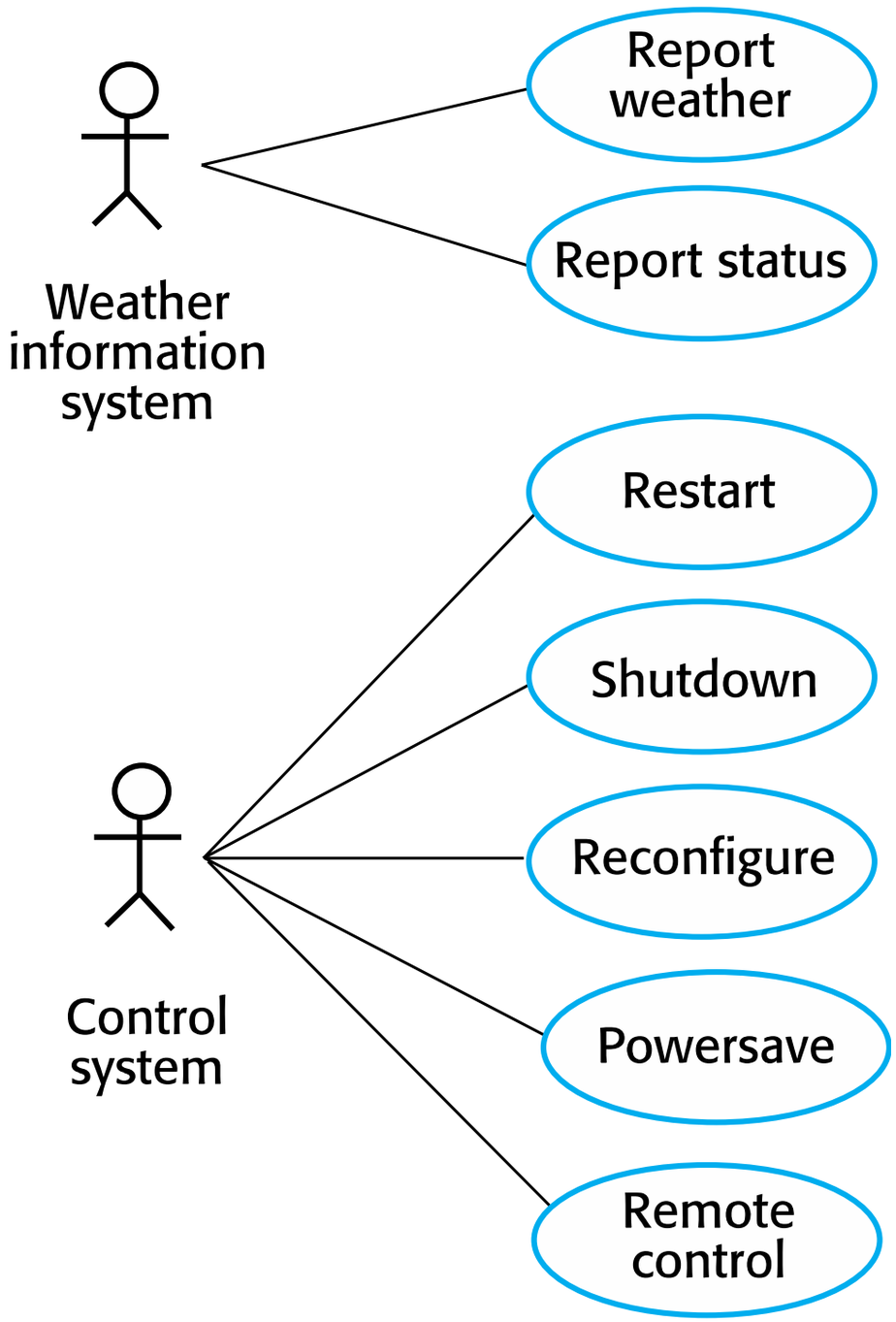
System context for the weather station

Context shown in class diagram



Weather station use cases

Interaction shown in use case diagram



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.

Object class identification

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

Approaches to identification

- Use a ***grammatical approach*** based on a natural language description of the system.

- ***Base the identification on tangible things*** in the application domain.

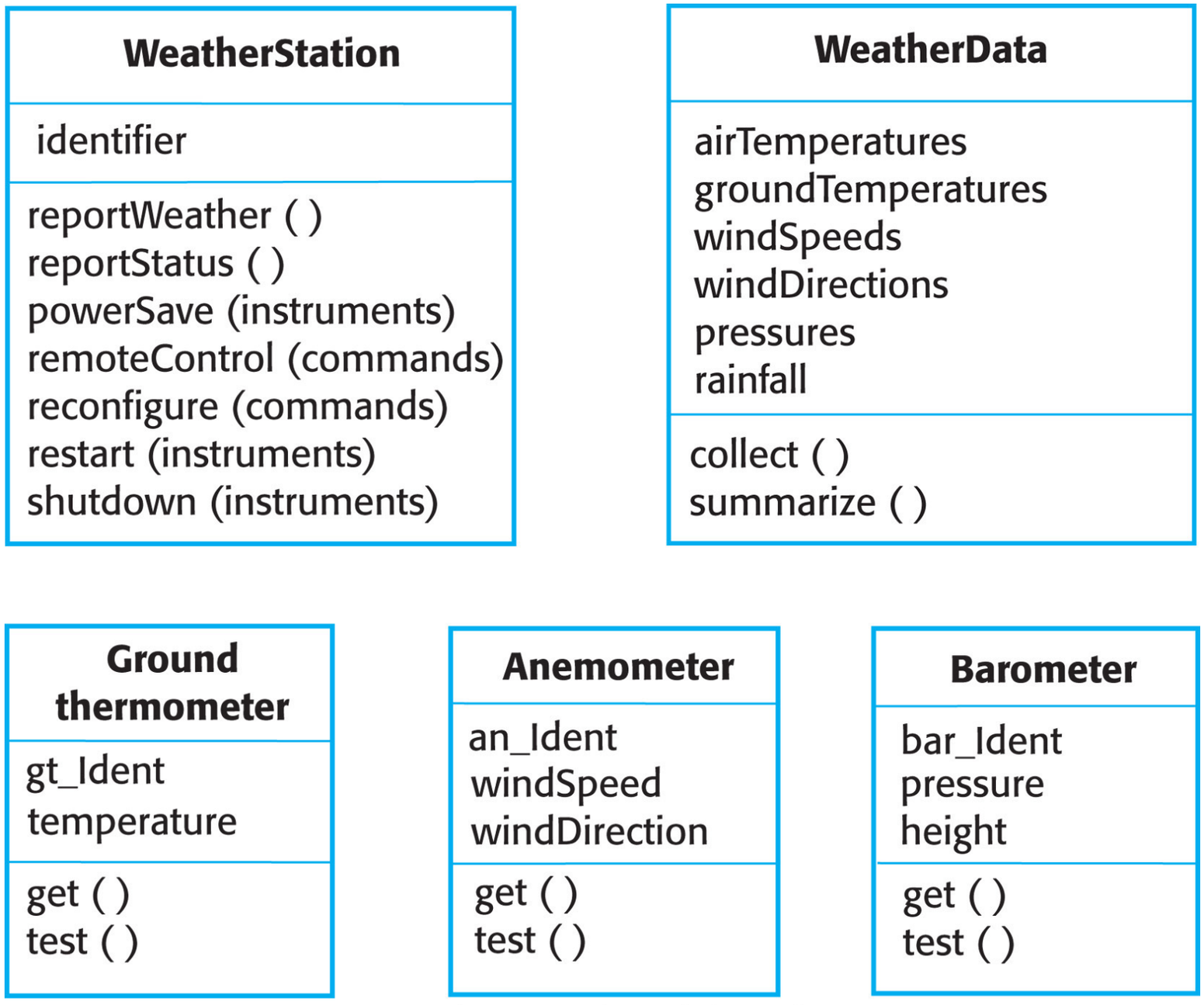
- Use a ***behavioural approach*** and identify objects based on what participates in what behaviour.

- Use a ***scenario-based analysis***. The objects, attributes and methods in each scenario are identified.

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



Design models

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

Examples of design models

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

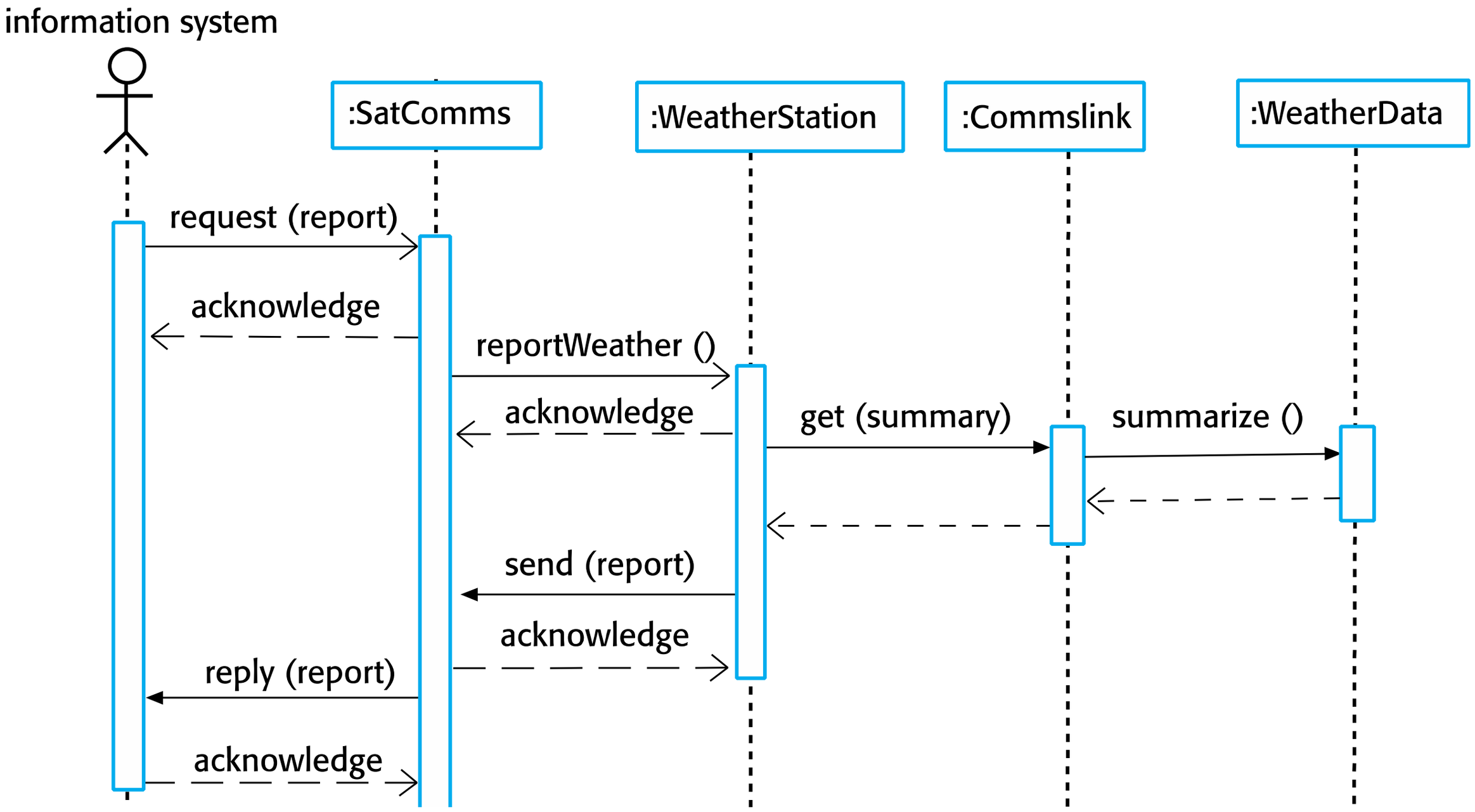
Subsystem models

- Shows how the design is organised into logically related groups of objects.
- In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

Sequence models

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

Sequence diagram describing data collection



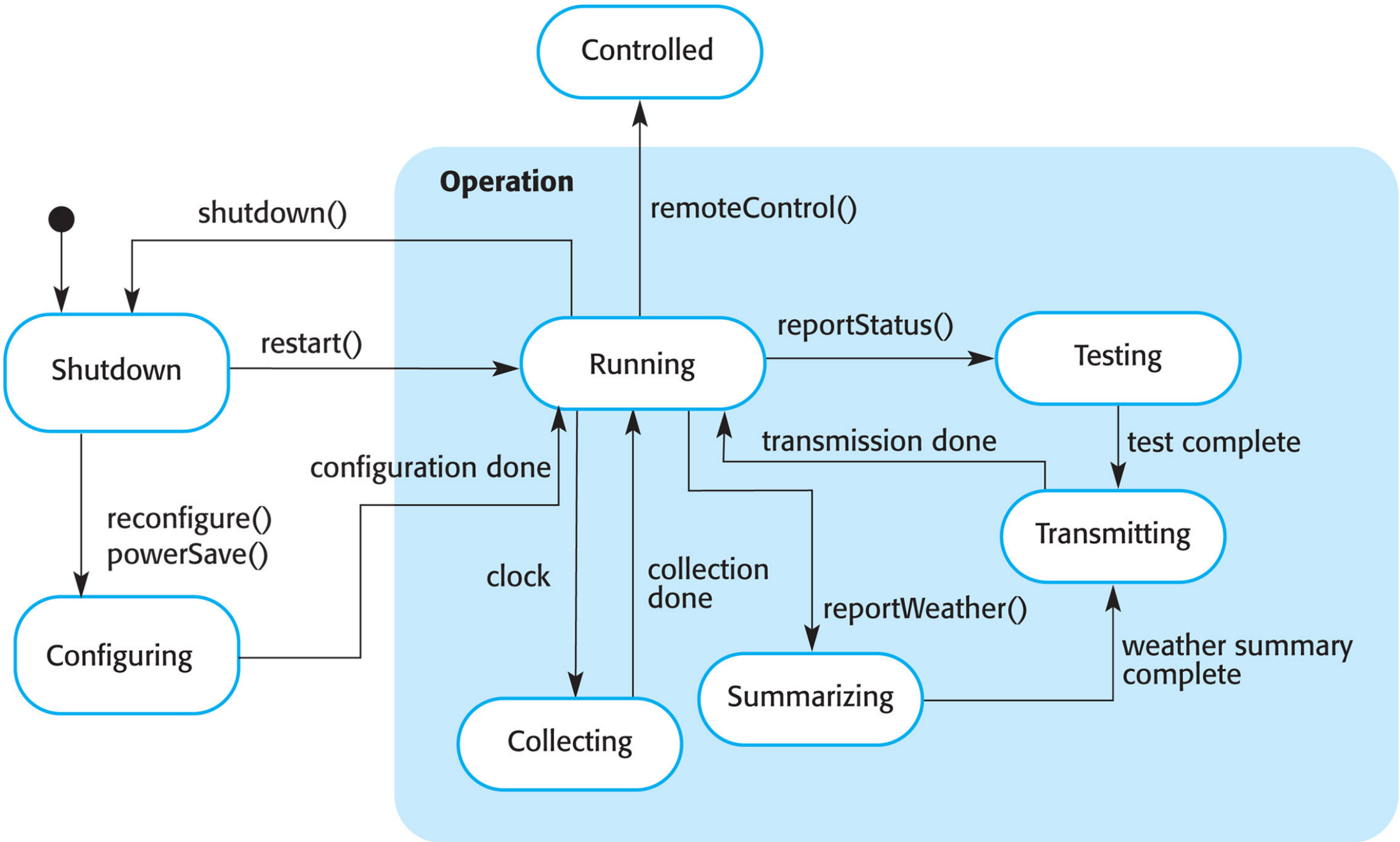
State diagrams

- State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.

- State diagrams are useful high-level models of a system or an object’s run-time behavior.

- You don’t usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

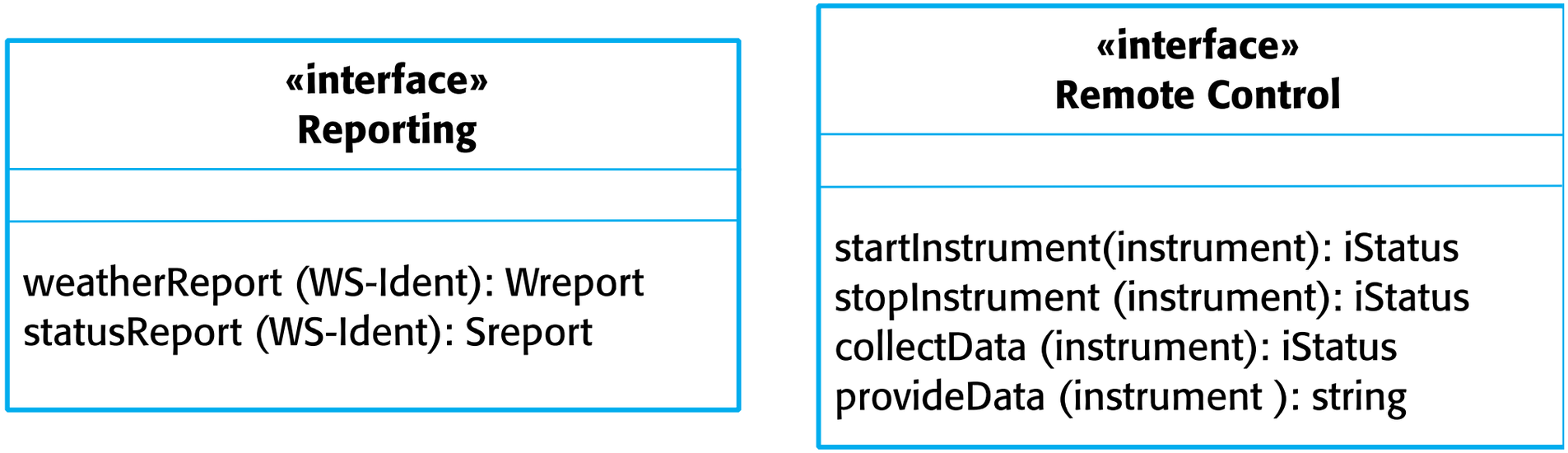
Weather station state diagram



Interface specification

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

Weather station interfaces



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.

Gang of Four (GoF)

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
- They wrote a book of a collection of design patterns.
- Design Patterns: Elements of Reusable Object-Oriented Software, 1995.
- Includes many design patterns about the design object-oriented software.

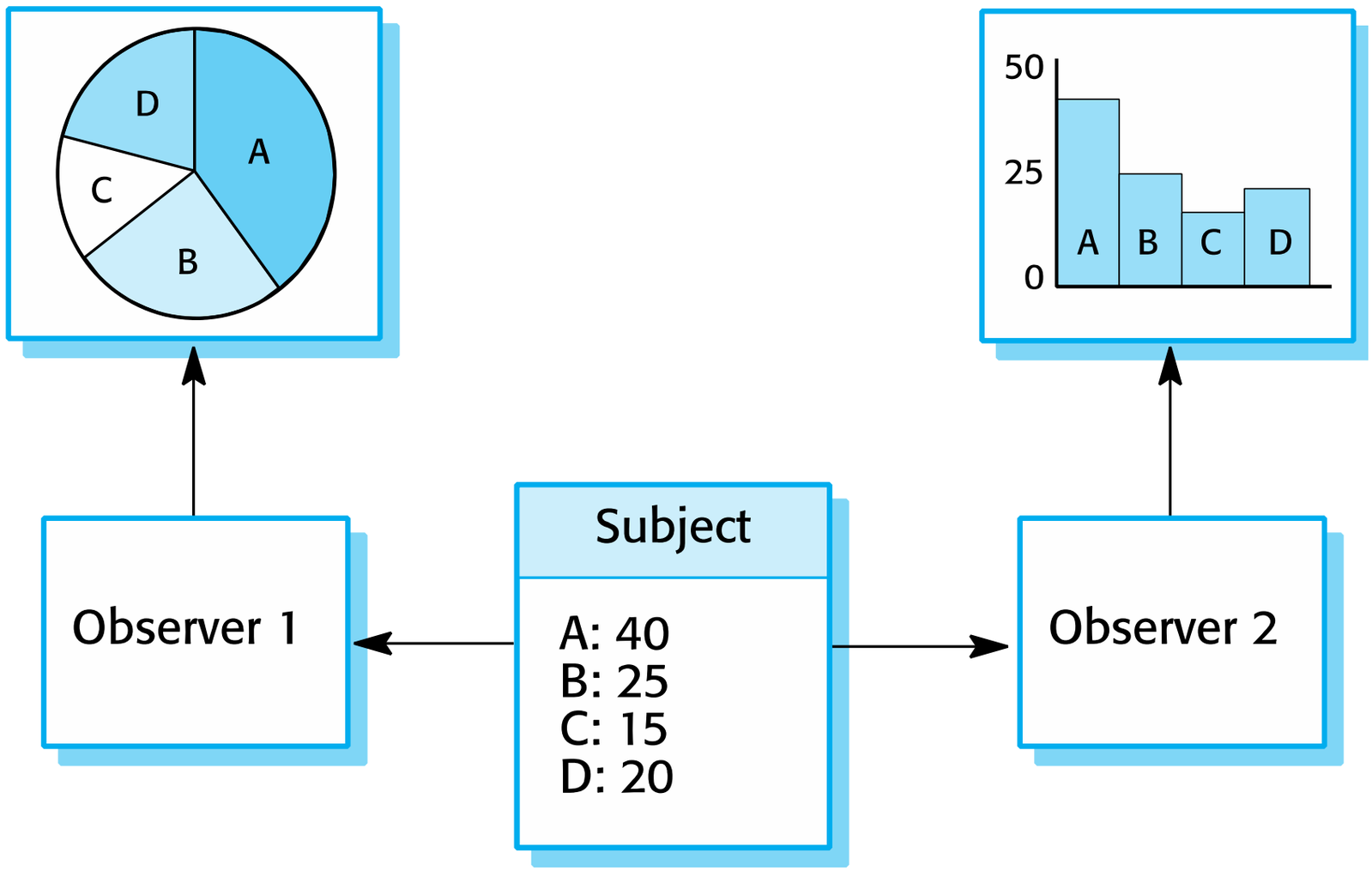
Pattern elements

- 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.

Multiple displays using the Observer pattern



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).
- Personally, I often use Adaptor, Visitor, Strategy and Factory method patterns.

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).

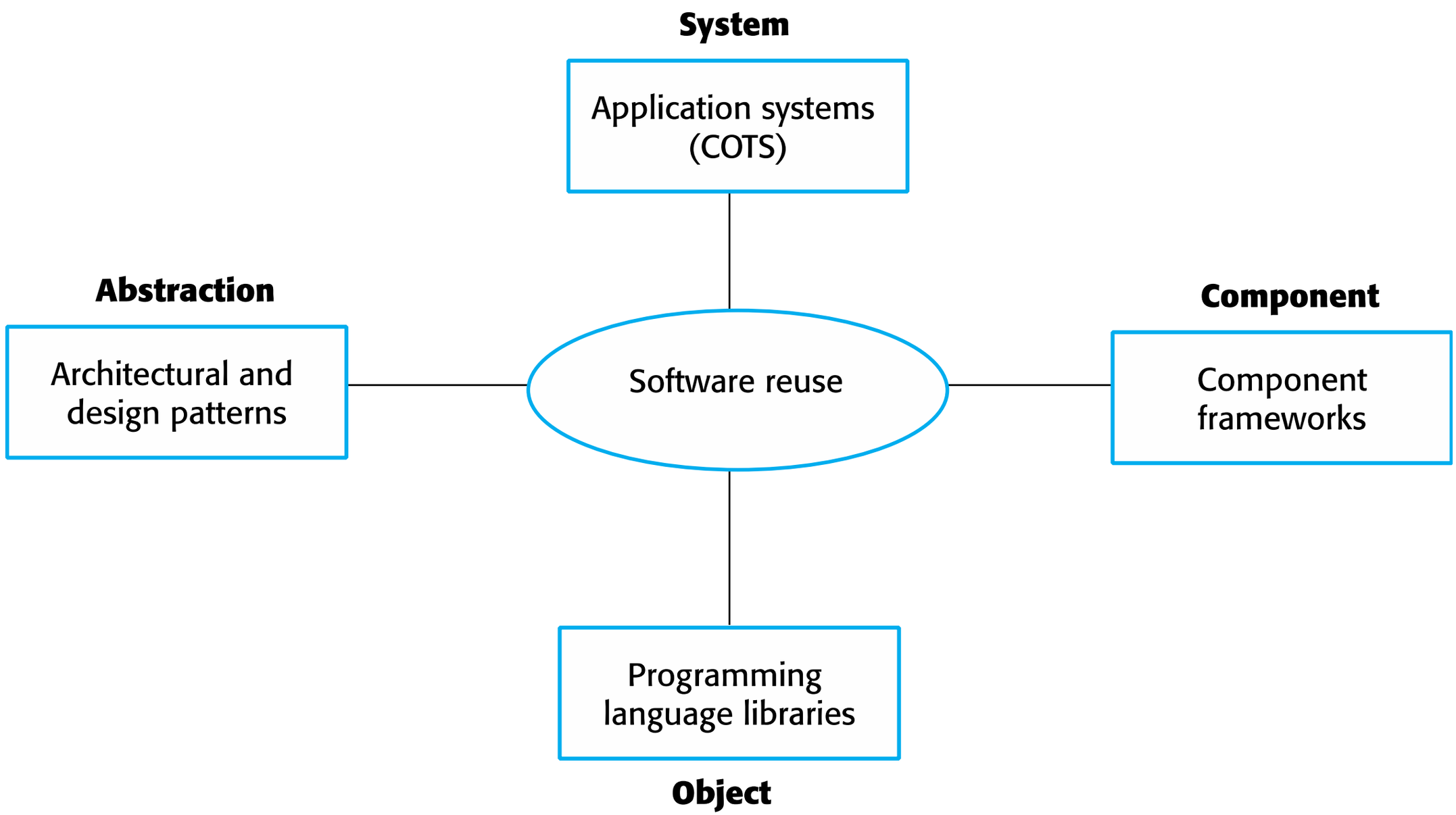
Reuse

- 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

- The component level
  - Components are collections of objects and object classes that you reuse in application systems.
- The system level
  - At this level, you reuse entire application systems.

Software reuse



Reuse costs

- The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.
- 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.
- See also Chapter 25.

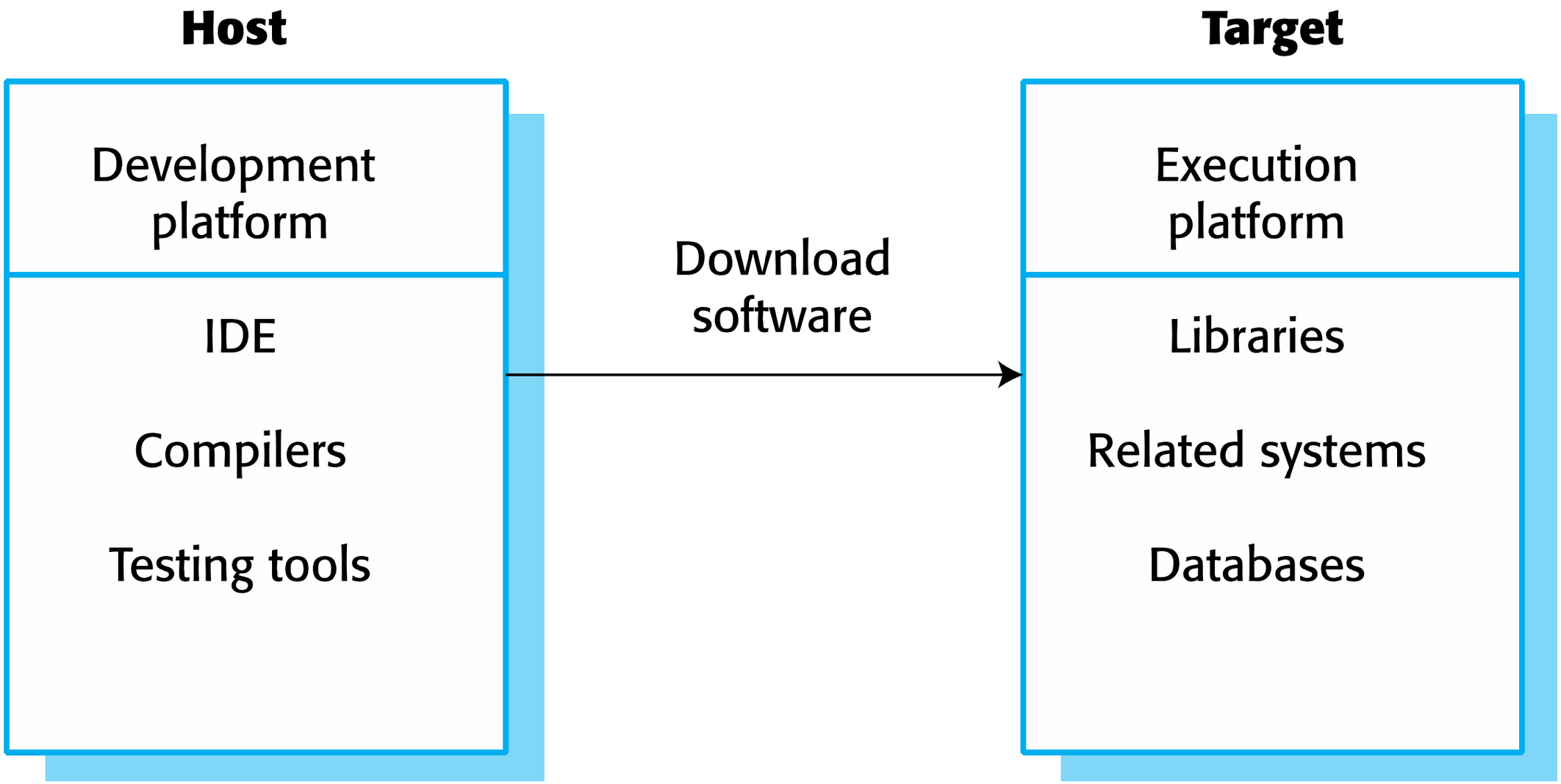
Configuration management activities

- **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.
- **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.
- **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.

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



Development platform tools

- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as Junit that can automatically run a set of tests on a new version of a program.
- Project support tools that help you organize the code for different development projects.

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.
    - This can be supported by new tools and techniques, such as cloud platform services like AWS or container services like Docker.
- 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.

Component/system deployment factors

- 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.
- For scalability, the same version of software may be deployed on multiple instances of platforms.
    - User traffic will be balanced on the instances, or even to instances with slightly different configurations.

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.

- Nowadays, open source development is very common, and even commercial software development is involved with open source software.

  - Recent dispute in Amazon and Elastic.

License models

- 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.

- 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.

- 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 models

- Github License Doc
- https://choosealicense.com/

Which of the following best describes your situation?

I need to work in a community.

Use the **license preferred by the community** you're contributing to or depending on. Your project will fit right in.

If you have a dependency that doesn't have a license, ask its maintainers to **add a license**.

I want it simple and permissive.

The **MIT License** is short and to the point. It lets people do almost anything they want with your project, like making and distributing closed source versions.

Babel, **.NET Core**, and **Rails** use the MIT License.

I care about sharing improvements.

The **GNU GPLv3** also lets people do almost anything they want with your project, *except* distributing closed source versions.

Ansible, **Bash**, and **GIMP** use the GNU GPLv3.

License models

GNU LGPLv3

Permissions of this copyleft license are conditioned on making available complete source code of licensed works and modifications under the same license or the GNU GPLv3. Copyright and license notices must be preserved. Contributors provide an express grant of patent rights. However, a larger work using the licensed work through interfaces provided by the licensed work may be distributed under different terms and without source code for the larger work.

Permissions

- Commercial use
- Distribution
- Modification
- Patent use
- Private use

Conditions

- Disclose source
- License and copyright notice
- Same license (library)
- State changes

Limitations

- Liability
- Warranty

View full GNU Lesser General Public License v3.0 »

Mozilla Public License 2.0

Permissions of this weak copyleft license are conditioned on making available source code of licensed files and modifications of those files under the same license (or in certain cases, one of the GNU licenses). Copyright and license notices must be preserved. Contributors provide an express grant of patent rights. However, a larger work using the licensed work may be distributed under different terms and without source code for files added in the larger work.

Permissions

- Commercial use
- Distribution
- Modification
- Patent use
- Private use

Conditions

- Disclose source
- License and copyright notice
- Same license (file)

Limitations

- Liability
- Trademark use
- Warranty

View full Mozilla Public License 2.0 »

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.

Importance of SE knowledge

- You might expect something more "practical" information for design and implementation.
- However, such things is not that important as you might think.
- Software development consists of highly intellectual and creative activities.
    - Hence it is more important to understand which things should be considered and executed during software development process.