MODULE 2

CHAPTER 2: DESIGN AND IMPLEMENTAION

2.1 Introduction to Rational Unified Process (RUP)

The Rational Unified Process (RUP) is an example of a modern process model that has been derived from work on the UML and the associated Unified Software Development Process.

The RUP is described from three perspectives:

1. A dynamic perspective, which shows the phases of the model over time.
2. A static perspective, which shows the process activities that are enacted.
3. A practice perspective, which suggests good practices to be used during the process.

Phases in the Rational Unified Process (A dynamic perspective)

The RUP is a phased model that identifies four discrete phases in the software process. The phases in the RUP are more closely related to business rather than technical concerns.

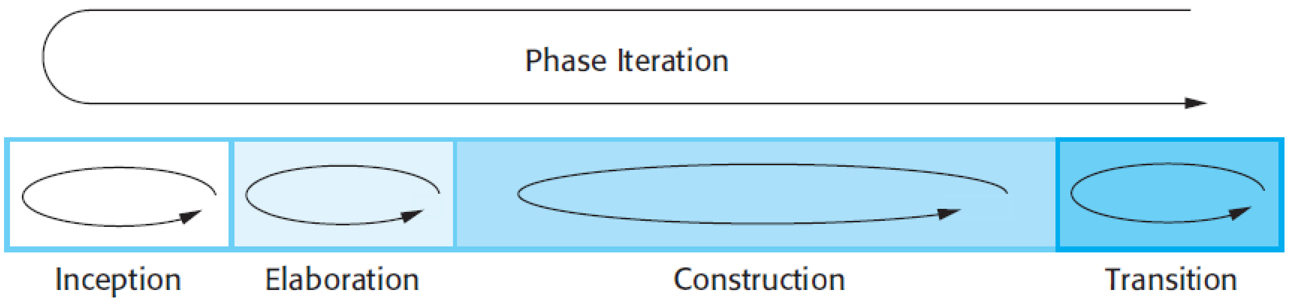


Figure: Phases in the Rational Unified Process

1. **Inception:** The goal of the inception phase is to establish a business case for the system. Here identify all external entities (people and systems) that will interact with the system and define these interactions. Then use this information to assess the contribution that the system makes to the business.

2. **Elaboration**: The goals of the elaboration phase are to develop an understanding of the problem domain, establish an architectural framework for the system, develop the project plan, and identify key project risks. On completion of this phase you should have a requirements model for the system, which may be a set of UML use-cases, an architectural description, and a development plan for the software.

3. **Construction**: The construction phase involves system design, programming, and testing. Parts of the system are developed in parallel and integrated during this phase. On completion of this phase, you should have a working software system and associated documentation that is ready for delivery to users.

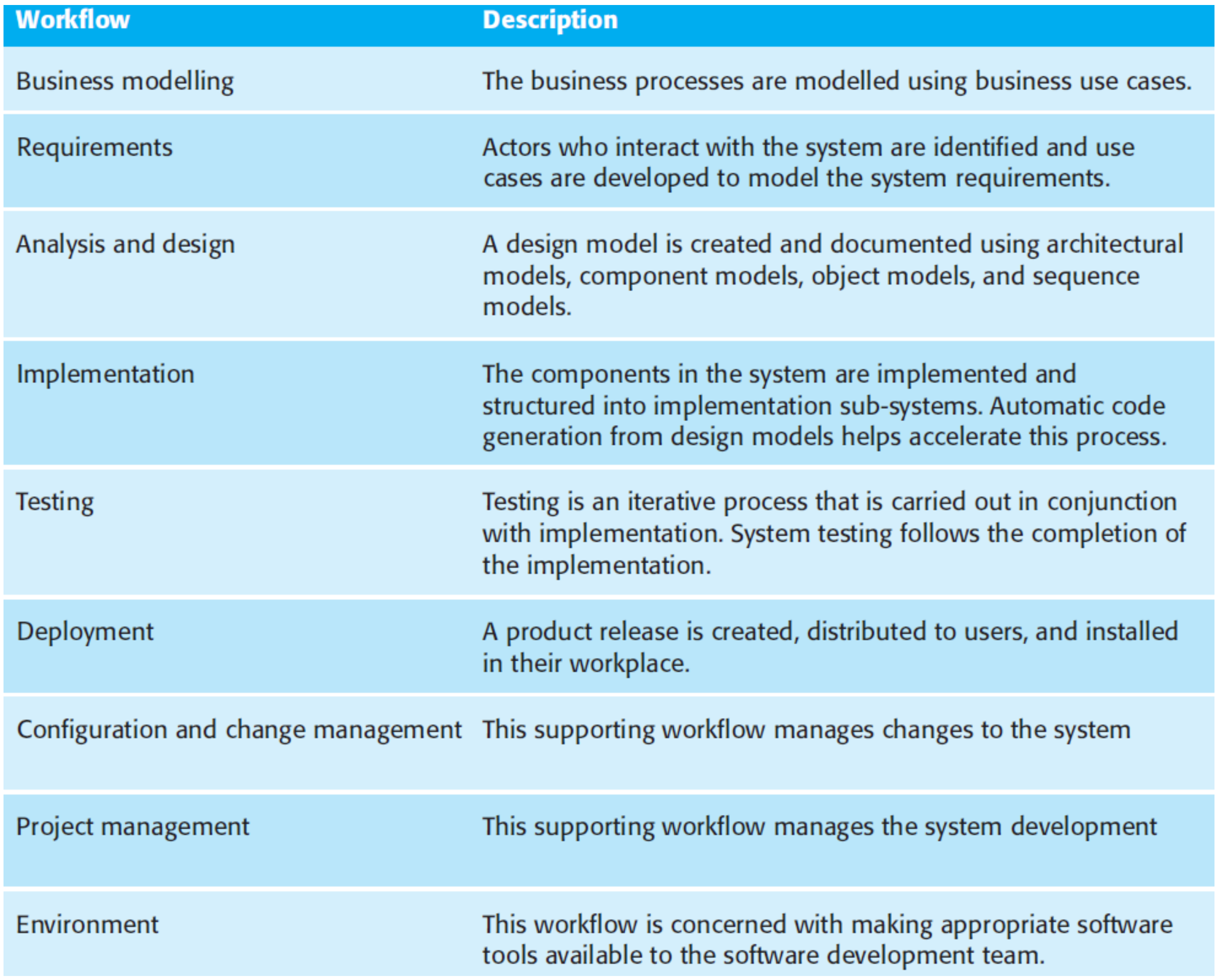
4. **Transition**: The final phase of the RUP is concerned with moving the system from the development community to the user community and making it work in a real environment. On completion of this phase, you should have a documented software system that is working correctly in its operational environment

Iteration within the RUP is supported in two ways.

1. Each phase may be enacted in an iterative way with the results developed incrementally.
2. The whole set of phases may also be enacted incrementally

Process activities in the Rational Unified Process (A static perspective)

- The static view of the RUP focuses on the activities that take place during the development process. These are called workflows in the RUP description.
- There are six core process workflows identified in the process and three core supporting workflows.



Workflow	Description
Business modelling	The business processes are modelled using business use cases.
Requirements	Actors who interact with the system are identified and use cases are developed to model the system requirements.
Analysis and design	A design model is created and documented using architectural models, component models, object models, and sequence models.
Implementation	The components in the system are implemented and structured into implementation sub-systems. Automatic code generation from design models helps accelerate this process.
Testing	Testing is an iterative process that is carried out in conjunction with implementation. System testing follows the completion of the implementation.
Deployment	A product release is created, distributed to users, and installed in their workplace.
Configuration and change management	This supporting workflow manages changes to the system
Project management	This supporting workflow manages the system development
Environment	This workflow is concerned with making appropriate software tools available to the software development team.

Figure: Static workflows in the Rational Unified Process

- The RUP has been designed in conjunction with the UML, so the workflow description is oriented around associated UML models such as sequence models, object models,

A practice perspective

The practice perspective on the RUP describes good software engineering practices that are recommended for use in systems development.

Six fundamental best practices are recommended:

1. **Develop software**: Iteratively plan increments of the system based on customer priorities and develop the highest-priority system features early in the development process
2. **Manage requirements**: Explicitly document the customer’s requirements and keep track of changes to these requirements. Analyze the impact of changes on the system before accepting them.
3. **Use component-based architectures**: Structure the system architecture into components
4. **Visually model software**: Use graphical UML models to present static and dynamic views of the software
5. **Verify software quality**: Ensure that the software meets the organizational quality standards
6. **Control changes to software**: Manage changes to the software using a change management system and configuration management procedures and tools.

Design Principles

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 interleaved.
- Software design is a creative activity in which software components and their relationships are identified, based on a customer's requirements.
- Implementation is the process of realizing the design as a program.

2.2 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.
- Object-oriented systems are easier to change than systems developed using functional approaches.
- To develop a system design from concept to detailed, object-oriented design, there are several activities need to do:
    1. Understand and define the context and the external interactions with the system.
    2. Design the system architecture.
    3. Identify the principal objects in the system.
    4. Develop design models.
    5. Specify interfaces.

2.2.1 System context and interactions

- The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment.
- 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.

- 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.
- Fig 2.1 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 one control system but several weather stations, one satellite, and one general weather information system.

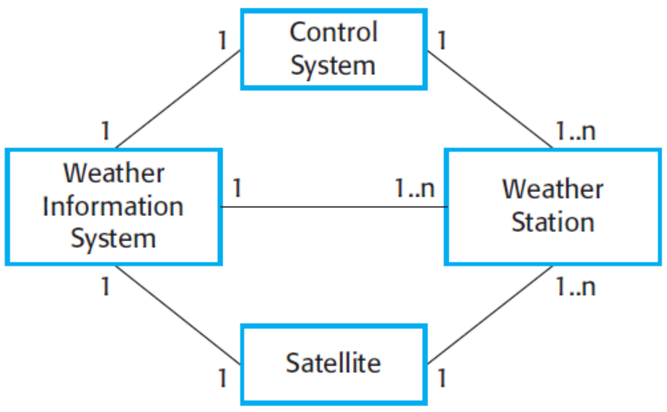


Figure 2.1: System context for the weather station

- The use case model for the weather station is shown in Figure 2.2. This 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.

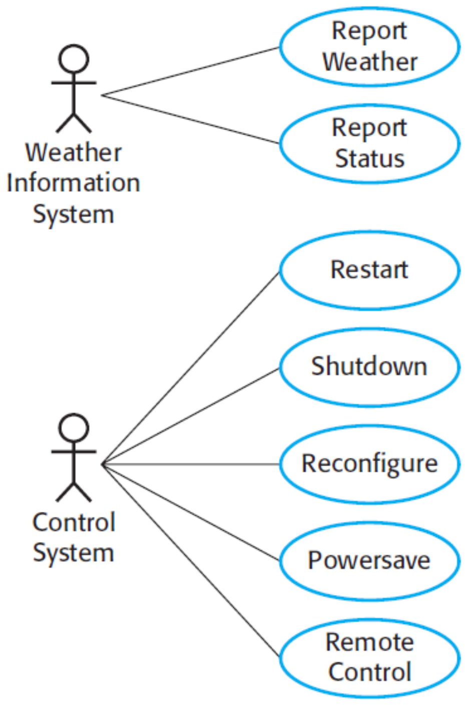
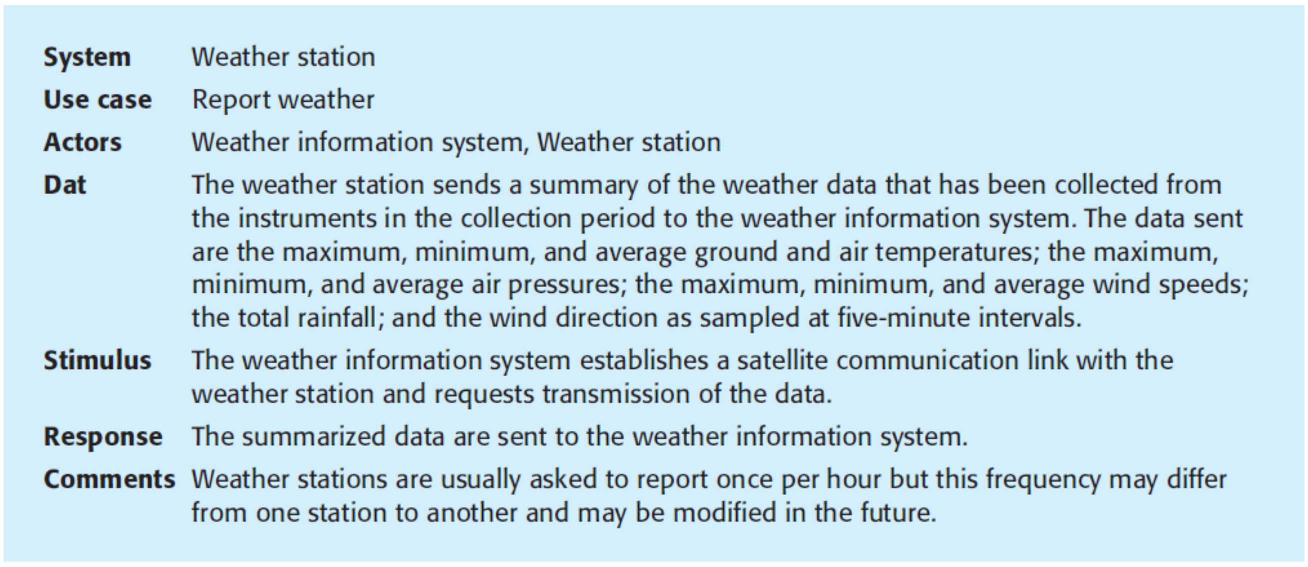


Figure 2.2: Weather Control station use cases

- Each of these use cases should be described in structured natural language. This helps designers to identify objects in the system and gives them an understanding of what the system is intended to do. Figure 2.3, which describes the Report weather use case from Figure 2.2.



System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Dat	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 are 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.

Figure 2.3: Use case description – Report weather

<u>2.2.2 Architectural design</u>

- The interactions between the software system and the system's environment is the basis for designing the system architecture.
- 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 high-level architectural design for the weather station software is shown in Figure 2.4.

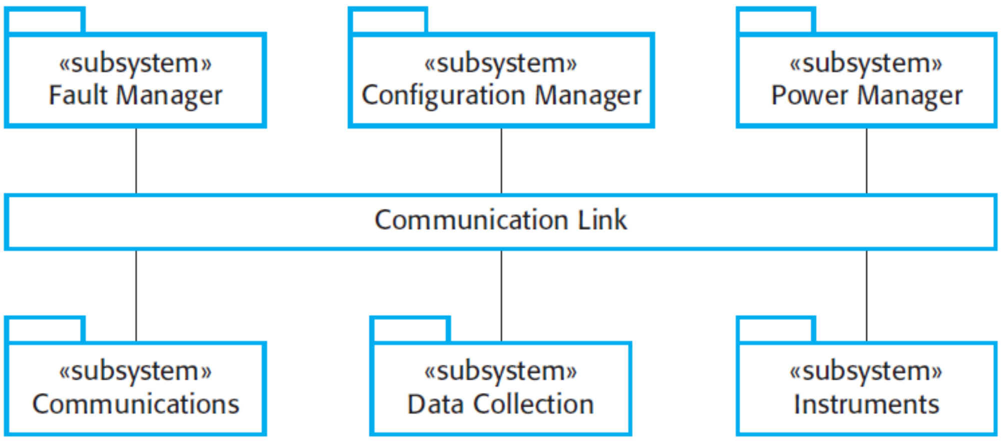

«subsystem» Fault Manager
«subsystem» Configuration Manager
«subsystem» Power Manager
Communication Link
«subsystem» Communications
«subsystem» Data Collection
«subsystem» Instruments

Figure 2.4: High-level architecture of the weather station

- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure, shown as the Communication link in Figure 2.4. Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them.
- For example, when the communications subsystem receives a control command, such as shutdown, the command is picked up by each of the other subsystems, which then shut themselves down in the correct way.
- The key benefit of this architecture is that 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.

Figure 2.5 shows the architecture of the data collection subsystem, which is included in Figure 2.4.

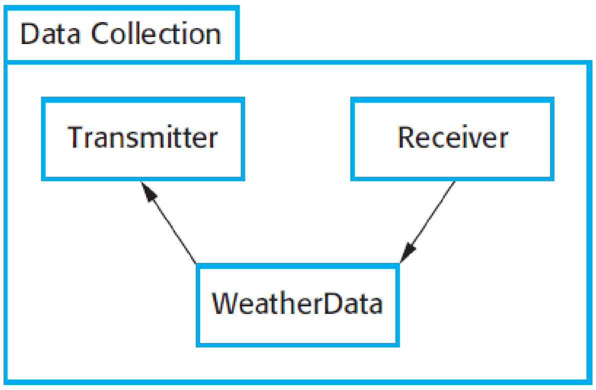


Figure 2.5: Architecture WeatherData of data collection system

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

2.2.3 Object class identification

Object identification is an iterative process. Various proposals are made to identify object classes in object-oriented systems

1. **Use a grammatical analysis** approach based on a natural language description of the system. Example: 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 or doctor, events such as requests, interactions such as meetings, locations such as offices, organizational units.
3. **Use a scenario-based analysis** where various scenarios of system use are identified and analysed and identify the required objects, attributes, and operations

- In the wilderness weather station, object identification is based on the tangible hardware in the system.

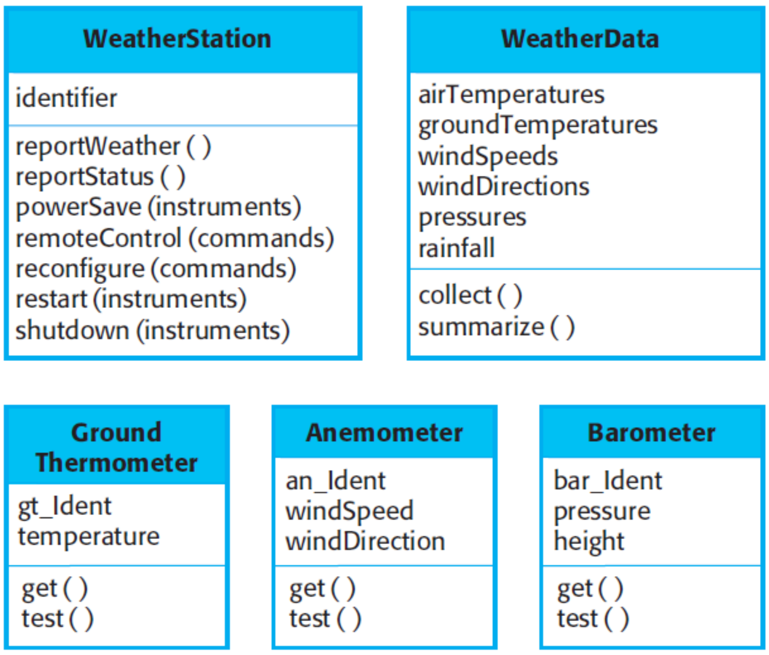


Figure 2.6: Weather station objects

- In figure 2.6 there are five object classes. 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.

2.2.4 Design models

Design or system models show the objects or object classes in a system. They also show the associations and relationships between these entities. These models are the bridge between the system requirements and the implementation of a system

When the UML is used to develop a design, two kinds of design models are normally developed:

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 interactions between the system objects.

The three models that are particularly useful for adding detail to use case and architectural models:

1. **Subsystem models** 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 static (structural) models.
2. **Sequence models** 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** show how individual objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models. State diagrams are useful high-level models of a system or an object's run-time behavior

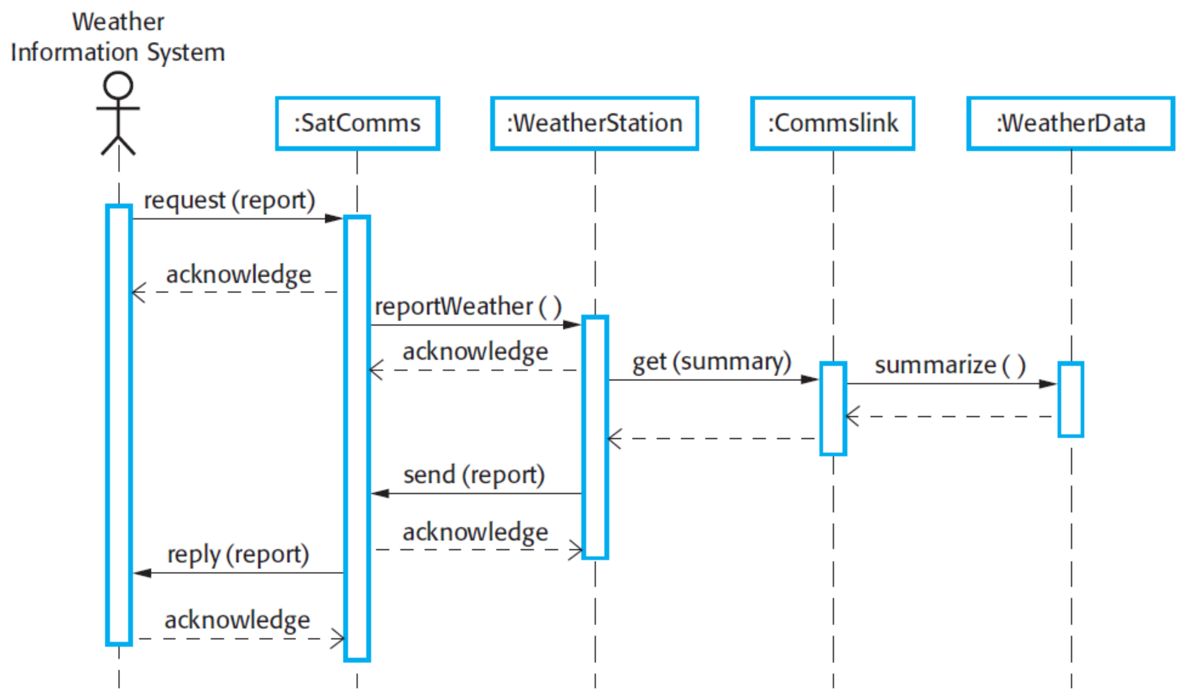


Figure 2.7: Sequence diagram describing data collection

Figure 2.7 is an example of a sequence model, shown as a UML sequence diagram. This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station.

Sequence diagrams are read from top to bottom:

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.
2. SatComms sends a message to WeatherStation, via a satellite link, to create a summary of the collected weather data.
3. WeatherStation sends a message to a Commslink object to summarize the weather data.

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.

Figure 7.8 is a state diagram for the weather station system that shows how it responds to requests for various services.

1. If the system state is Shutdown then it can respond to a restart(), a reconfigure(), or a powerSave() message.
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.
4. If a reportStatus() message is received, the system moves to the Testing state, then the Transmitting state, before returning to the Running state.
5. 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.

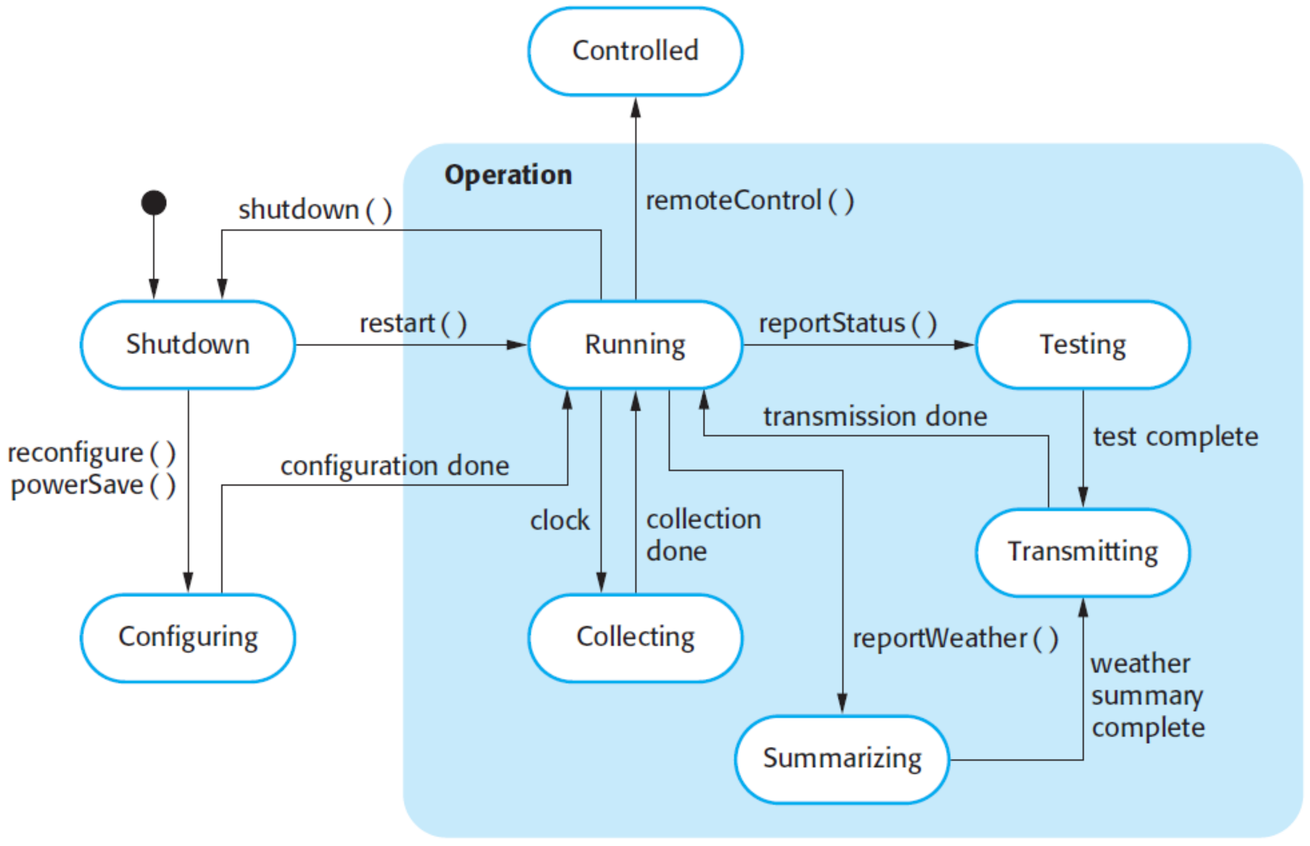


Figure 2.8: Weather station state diagram

2.2.5 Interface specification

- Interface design is concerned with specifying the detail of the interface to an object or to a group of objects
- Interfaces can be specified in the UML using the same notation as a class diagram. There is no attribute section and the UML stereotype ‹‹interface›› should be included in the name part.
- Figure 2.9 shows two interfaces that may be defined for the weather station. The reporting interface 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.

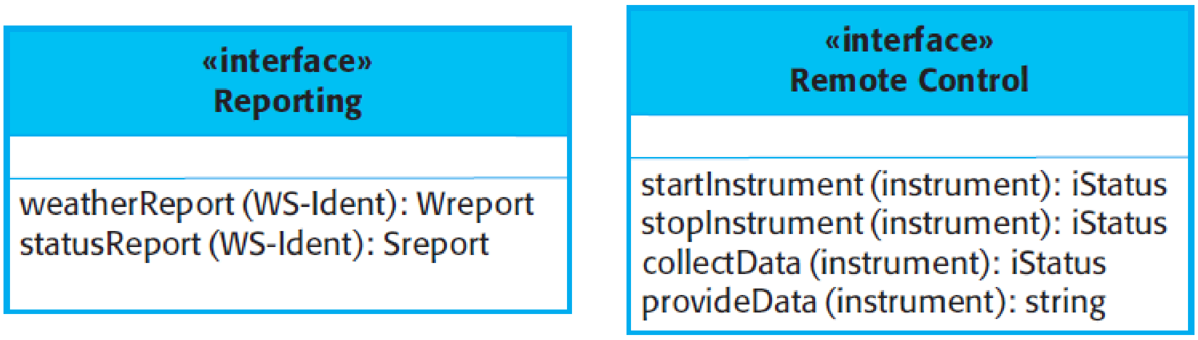

«interface»
Reporting
weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport
«interface»
Remote Control
startInstrument (instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument): string

Figure 2.9: Weather station interfaces

2.3 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.

The 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. This is not a concrete design description. 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.

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.

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.

The UML model of the pattern is shown in Figure 7.12.

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.

Figure 2.10: The Observer pattern

Figure 2.10 to illustrate pattern description. There are four essential description elements and also include a brief statement of what the pattern can do. This pattern can be used in situations where different presentations of an object's state are required.

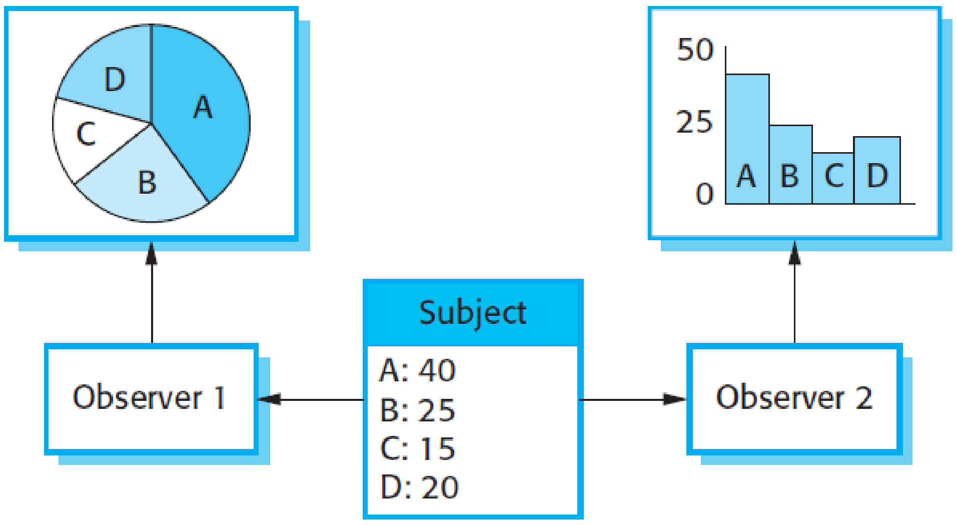


Figure 2.11: Multiple displays

Figure 2.11, which shows two graphical presentations of the same data set.

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. Figure 2.12 is the representation in UML of the Observer pattern

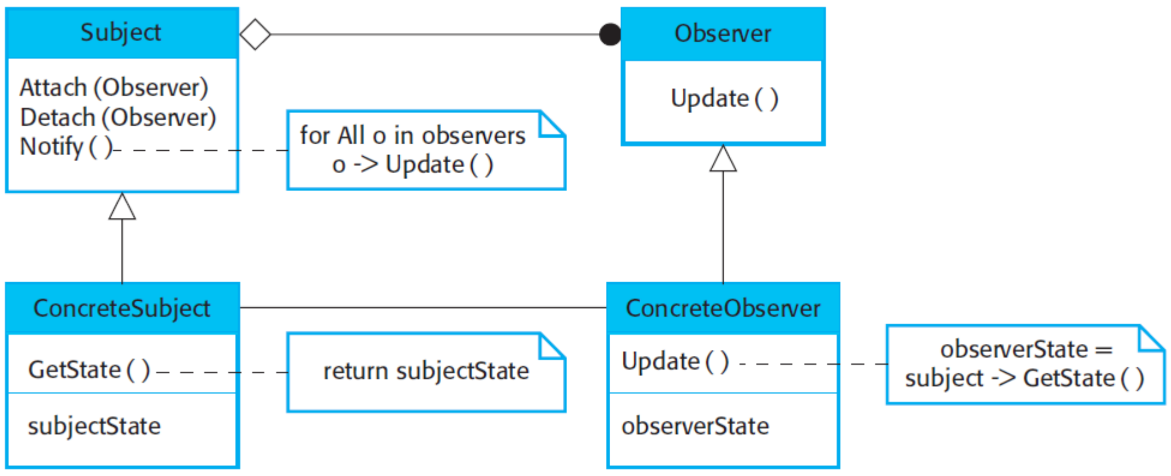


Figure 2.12: A UML model of the Observer pattern

2.4 Implementation issues

Implementation is a critical stage where an executable version of the software is developed.

Some of the aspects of implementation are:

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

2.4.1 Reuse

The reuse of existing software has emerged and is now generally used for business systems, scientific software, and, increasingly, in embedded systems engineering.

Software reuse is possible at a number of different levels:

1. **The abstraction level:** At this level, software is not reused directly but rather use knowledge of successful abstractions in the design of your software.

2. **<u>The object level:</u>** At this level, directly reuse objects from a library rather than writing the code yourself. For example, to process mail messages in a Java program, use objects and methods from a JavaMail library.
3. **<u>The component level:</u>** Components are collections of objects and object classes that operate together to provide related functions and services. Adapt and extend the component by adding some code of your own. An example of component-level reuse is where building user interface using a framework.
4. **<u>The system level:</u>** At this level, reuse entire application systems.

By reusing existing software, develop new systems can be developed more quickly, with fewer development risks and also lower costs. However, there are 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-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 and with the new code which is developed.

<u>2.4.2 Configuration management</u>

- 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.

<u>There are three fundamental configuration management activities:</u>

1. **<u>Version management:</u>** 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.
2. **<u>System integration:</u>** 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. **<u>Problem tracking:</u>** 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.

2.4.3 Host-target development

- Most software is developed on one computer (the host, development platform), but runs on a separate machine (the target, 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 (IDE). An IDE is a set of software tools that supports different aspects of software development, within some common framework and user interface.
- Development platform usually has different installed software than execution platform; these platforms may have different architectures. Mobile app development (e.g. for Android) is a good example.

Typical development platform tools 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. Project support tools that help you organize the code for different development projects.

2.5 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, which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish.
- 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.
- 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.

Typical licensing models include:

- **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.