Chapter 7
Design and Implementation
Topics covered

- Object-oriented design using the UML
- Design patterns
- Implementation issues
  - Reuse
  - Configuration management
  - Host-target development
- Open source development
  - Development issues and business model
  - License models
What is “design and implementation”? 

- The stage in the software engineering process at which an executable software system is developed.
- Design and implementation activities are invariably interleaved.
  - **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 one or more executable programs.
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 users’ requirements.
  - For example, if a medical records system is required, you can buy a package that is already used in hospitals. This can be cheaper and faster than developing a system.

- When you procure 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, and for small systems, this may not be cost-effective.*
- However, for large systems developed by different groups, system models are an important communication mechanism.*

* acknowledgment of both agile and planned-based principles
...an iterative, boot-strapping process
Process stages

- There are a variety of different object-oriented design processes.
- But the common activities normally include:
  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 (static and dynamic).
  5. Specify object interfaces.
- The process illustrated here is for the wilderness weather station. (See Section 1.3.3, p. 22.)
System context and interactions

- **Goal:** develop an understanding of the relationships between the software being designed and its external environment.
- Understanding of the system context also lets you establish the boundaries (scope) of the system.
- This helps in deciding what features will be implemented in the system being designed and what features will be in associated systems.
Context and interaction models

- A system *context model* is a *structural model* that identifies 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.
Context model for the weather station

![Diagram showing relationships between control system, weather information system, weather station, and satellite with cardinality information.]
Interaction model based on Use Cases

- Weather information system
  - Report weather
  - Report status

- Control system
  - Restart
  - Shutdown
  - Reconfigure
  - Powersave
  - Remote control
Use case description—Report weather

<table>
<thead>
<tr>
<th>System</th>
<th>Weather station</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case</td>
<td>Report weather</td>
</tr>
<tr>
<td>Actors</td>
<td>Weather information system, Weather station</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
<tr>
<td>Stimulus</td>
<td>The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.</td>
</tr>
<tr>
<td>Response</td>
<td>The summarized data is sent to the weather information system.</td>
</tr>
<tr>
<td>Comments</td>
<td>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.</td>
</tr>
</tbody>
</table>
Architectural design

- Once interactions between the system and its environment have been understood, this info is used in designing the system architecture.

- Identify the major components that make up the system and their interactions, and then organize the components using an architectural pattern such as a layered or client-server model.

- The weather station is comprised of independent subsystems that communicate by broadcasting messages via a shared infrastructure (another common architectural style in addition to those described in Chapter 6)…
High-level architecture of weather station†

† OO sub-system design model using UML “packages”
Object class identification

- Identifying object classes is often a difficult part of OO design.
- There is no “magic formula” – it relies on the skill, experience, and domain knowledge of system designers.
- An iterative, boot-strapping process – you are unlikely to get it right the first time.
Approaches to object identification

- Use a grammatical approach based on a natural language description of the system (Abbott’s heuristic).
- Associate objects with tangible things in the application domain (e.g., devices).
- Use a behavioural approach: identify objects based on what participates in what behaviour.

(cont’d)
Approaches to object identification (cont.)

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

- Use an **information-hiding based approach**.*

  *Identify potentially changeable design decisions and isolate these in separate objects to minimize the impact of change.* (Parnas)

* “Bonus” approach! (No extra charge.)
Weather station description

A weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected every five minutes.
Weather station description (cont’d)

When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarized data is transmitted to the mapping computer when a request is received.
Weather station object classes

- Object class identification may be based on the tangible hardware and data in the system:
  - **Weather station** — interface of the weather station to its environment. It reflects interactions identified in the use-case model.
  - **Weather data** — encapsulates summarised data from the instruments.
  - **Ground thermometer, Anemometer, Barometer, etc.** — application domain “hardware” objects* related to the instruments in the system.

* hardware-controlling SOFTWARE
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()
Design models

- **OO design models** show the objects or object classes in a system, and their relationships.
  - *Structural models* describe the static structure of the system in terms of object and object class relationships.
  - *Dynamic models* describe the dynamic interactions among objects.
Examples of OO design models

- **Sub-system models** show logical groupings of objects into coherent sub-systems. *(structural)*
- **Sequence models** show the sequence of object interactions associated with system uses. *(dynamic)*
- **State machine models** show how individual objects change their state in response to events. *(dynamic)*
- Other models include use-case models, aggregation models, generalisation (inheritance) models, etc.
Sub-system models

- Show how the design is organized into logically related groups of objects.
- In the UML, these are shown using packages, an encapsulation construct.
- These are **logical models** – the actual organization of objects in a system as implemented may be different.
Weather station sub-systems

«subsystem»
Interface
CommsController
WeatherStation

«subsystem»
Data collection
WeatherData
Instrument
Status

«subsystem»
Instruments
Air
thermometer
RainGauge
Anemometer
Ground
thermometer
Barometer
Wind Vane

Active object
Annotations
go here
Sequence models

- Show the sequence of object interactions that take place.
- Objects are arranged horizontally across the top.
- Time is represented vertically; models are read top to bottom.
- Interactions are represented by labelled arrows – different styles of arrows represent different types of interaction.
- A thin rectangle in an object lifeline represents the time when the object is active.
Data collection sequence diagram
State diagrams

- 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 an object’s (or a system’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

by messages received from “remote control room”

initial state
Interface specification

- Object interfaces have to be specified to support concurrent design activity (among other things...).

- Designers should avoid revealing data (origin of “info hiding” term) representation information in their interface design. (operations access and update all data)

- Objects may have several logical interfaces which are viewpoints on the methods provided. (supported directly in Java)

- UML class diagrams are used for interface specification, but pseudocode may also be used.
Weather station interfaces

```plaintext
«interface» Reporting

weatherReport (WS-Ident): Wreport

«interface» Remote Control

startInstrument(instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument ): string
```
Design patterns
Design patterns

- A way of reusing "accumulated knowledge and wisdom" about a problem and its solution.
- A design pattern is a description of some problem and the "essence" of a solution.
- Should be sufficiently abstract to be reusable in different contexts.
- Often utilize OO characteristics such as inheritance and polymorphism.
Pattern elements

- **Name**: a meaningful *pattern identifier*
- **Problem description**
- **Solution description**: a template for a design solution that can be instantiated in different operational contexts *(often illustrated graphically)*
- **Consequences**: the results and trade-offs of applying the pattern *(analysis and experience)*
Example: The Observer pattern*

* cf Model-View-Controller (MVC) architectural design pattern
The Observer pattern

- **Name:** Observer
- **Description:** Separates the display of object state from the object itself allowing alternative displays.
- **Problem description:** Used when multiple displays of state are needed.
- **Solution description:** (See UML description.)
- **Consequences:** Object optimizations to enhance the performance of a particular display are impractical.

(cont’d)
### The Observer pattern (cont’d)

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Problem description</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
## The Observer pattern (cont’d)

<table>
<thead>
<tr>
<th>Pattern name</th>
<th>Observer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solution description</strong></td>
<td>This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObserver, 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.</td>
</tr>
<tr>
<td><strong>Consequences</strong></td>
<td>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.</td>
</tr>
</tbody>
</table>
The Observer pattern (cont’d)

- **Concrete Subject**
  - Has any number of observers
  - Provides an interface to attach and detach observer objects at run-time
  - Sends notification to its observers

- **Concrete Observer**
  - Provides an update interface to receive signals from subject
  - Implements update operation
UML model of Observer pattern

Subject super class

Observer super class (for alternative displays)

Specific subject sub-class

ConcreteSubject

GetState ()

subjectState

Specific observer sub-class

ConcreteObserver

Update ()

observerState

observerState = subject -> GetState ()

Subject

Attach (Observer)

Detach (Observer)

Notify ()

for all o in observers

o -> Update ()

Observer

Update ()

one to many
Recognizing design patterns

- Using patterns effectively requires the ability to recognize common design problems and their associated solution patterns.

- For example:
  - Tell several objects that the state of some other object has changed (Observer pattern).
  - Provide a standard way of accessing the elements in a collection sequentially, irrespective of how that collection is implemented (Iterator pattern).
  - Allow for the possibility of extending the functionality of an existing object at run-time (Decorator pattern).
Implementation issues
The focus here is NOT on programming…

- **Reuse**: Most modern software is constructed by reusing existing components or systems.

- **Configuration management**: Keeping track of the different versions of software components using a configuration management system.

- **Host-target development**: Production software is usually developed on one computer (the host system) and executes on a separate computer (the target system).
The focus here is NOT on programming…

- **Reuse**: Most modern software is constructed by reusing existing components or systems.

- **Configuration management**: Keeping track of the different versions of software components using a configuration management system.

- **Host-target development**: Production software is usually developed on one computer (the host system) and executes 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 of software was the reuse of functions and procedures in programming language libraries.

- Costs and schedule pressure have made this approach increasingly impractical, especially for commercial and Internet-based systems.

(cont’d)
Reuse (cont’d)

- Development based around the reuse of existing software therefore emerged and is now generally used for most business and scientific software.
Reuse levels

- **The abstraction level**: reuse knowledge of successful abstractions (e.g., patterns) in the design of your software.

- **The object level**: reuse objects from a library rather than writing the code yourself.

- **The component level**: components are collections of objects and object classes that may be reused in application systems.

- **The system level**: reuse entire application systems.
Reuse costs include...

- The time spent in looking for software to reuse and assessing whether or not it meets the 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 reusable software components or systems to reflect the requirements of the system being developing.
- The costs of integrating reusable software elements with each other (when using software from different sources) and with any new code being developed.
The focus here is NOT on programming…

- **Reuse**: Most modern software is constructed by reusing existing components or systems.

- **Configuration management**: Keeping track of the different versions of software components using a configuration management system.

- **Host-target development**: Production software is usually developed on one computer (the host system) and executes on a separate computer (the target system).
Configuration management

- Configuration management is the name given to the **general process of managing a changing software system**.
- Its aim is to support the system integration process so that all developers can (1) access the project code and documents in a controlled way, (2) find out what changes have been made, and (3) compile and link components to create (“build”) a system.
Configuration management activities

- **Version management**: keeping track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.

- **System integration**: defining what versions of components are used to create each version of a system. Used to build a system automatically by compiling and linking the required components.

- **Problem tracking**: allows reporting of bugs and other problems, and allows all developers to see who is working on these problems and when they are fixed.
The focus here is NOT on programming…

- **Reuse**: Most modern software is constructed by reusing existing components or systems.

- **Configuration management**: Keeping track of the different versions of software components using a configuration management system.

- **Host-target development**: Production software is usually developed on one computer (the host system) and executes on a separate computer (the target system).
Host-target development

- Most software is developed on one computer (the host or development platform), but runs on a separate machine (the target or execution platform).
- A platform is more than just hardware; it includes the installed OS plus other supporting software such as a DBMS or, for development platforms, an Integrated Development Environment (IDE).
- The development platform usually has different installed software than the execution platform and may have a different architecture.
Development platform tools

- An integrated compiler and syntax-directed editing system to create, edit, and compile code.
- A language-specific 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) bundle these tools...

- An IDE is an *integrated* set of software tools on a host machine that supports different aspects of software development.
- IDEs are created to support development in a specific programming language such as Java.
Deployment (execution) platform issues (for distributed systems)

- Components must obviously be deployed on a platform that provides the hardware and software support they require.
- High availability systems may require deployment on more than one platform. (In the event of platform failure, an alternative implementation of the component is available.)
- If the communication traffic between components is heavy, it usually makes sense to deploy them on the same platform or on platforms that are physically close to one other.
Open source development
Open source development

- 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. Instead, it should always be available for users to examine and modify as they wish.

(cont’d)
Open source development (cont’d)

- “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.
- The Linux operating system is probably the best known open source product.
- Other important open source products are Java, the Apache web server, and the mySQL database management system.
Open source development issues and business model

- **Should a product that is being developed make use of (existing) open source components?**
  - Are there high-quality open source components available? (Is the product general or specific to some organization?)
  - Are time to market and reduced cost critical?

- **Should an open source approach be used when developing software?**
  - Is volunteer involvement likely?
  - Is a business model based on selling support for a product (rather than selling the product itself) appropriate?
Open source licensing

- Making source code freely available does not mean that anyone can do as they wish with that code.
- The developer still owns the code and can place restrictions on how it is used by including legally binding conditions in an open source software license.

(cont’d)
Open source licensing (cont’d)

- 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 - i.e., the developed systems may be proprietary and sold as closed source systems.
General OS license models

- The GNU General Public License (GPL): 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): 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.

(cont’d)
The Berkley Standard Distribution (BSD) License: 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.
Open Source Initiative (OSI)

- The OSI is the main public advocacy organization promoting non-proprietary software.
- There are MANY specific OSI-approved licenses in use today.
- All must comply with the Open Source Definition.
- See opensource.org for a current list.
- Also, see “Relationship with the free software movement” here.
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.

(cont’d)
Key points (cont’d)

- 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. UML may be used to define interfaces.

(cont’d)
Key points (cont’d)

- 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.

(cont’d)
Key points (cont’d)

- Most software development is based on a host-target model. 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.
Chapter 7

Design and Implementation