Design and Implementation

Chapter 7

Design and Implementation
in the textbook

- Introduction
- 7.1 Object-oriented design using the UML
- 7.2 Design patterns
- 7.3 Implementation issues
- 7.4 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 interleaved.
- Design should be related to the implementation environment
  - Don't use UML (object-oriented design) to write code in C.

Design and Implementation

- Software design:
  - Creative activity
  - Identify software components and their relationships
  - Based on requirements.
- Implementation is the process of realizing the design as a program.
- Design may be
  - Documented in UML (or other) models
  - Informal sketches (whiteboard, paper)
  - In the programmer's head.

Purpose of the chapter

- It is NOT about programming topics
  - Assume you all have design and implementation experience.
- To show how system modeling (ch 5) and architectural design (ch 6) are practiced in object oriented design
- To introduce implementation issues not usually covered in programming books
  - Software reuse
  - configuration management
  - open source development

7.1 Object-oriented design using UML

- Object-oriented system made up of interacting objects
  - Maintain their own local state (private).
  - Provide operations over that state.
- Object-oriented design process:
  - Design classes (for objects) and their interactions.
- Why object oriented?
  - Data is encapsulated: can change representation without changing code external to class.
  - can add services without affecting other classes.
  - clear mapping between classes and real world objects.

Object-oriented design activities

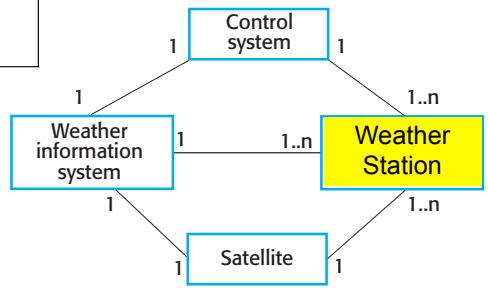
- Main activities:
  - Understand and define context and external interactions with the system
  - Design system architecture
  - Identify the principal objects in the system
  - Develop design models.
  - Specify interfaces
- Get ideas, propose solutions, refine, trial and error, backtrack, explore options, defer details...
- Illustrate process by designing wilderness weather station.

7.1.1 System context and interactions

- First: understand the relationship between the software being developed and its external environment.
  - sets boundaries: know what to implement
  - decide how to structure system so it can communicate with its environment
- System models to use here:
  - Context model: Boxes and lines (+ cardinality), --shows all systems involved
  - Interaction models: Use cases (or UML activity diagram) --shows how the systems interact

System context for weather station

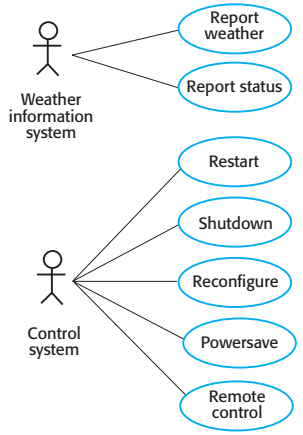
one Weather information system,
one Control system,
one Satellite,
many Weather stations.



We are designing the Weather station system.

Weather station use cases

How the Weather information system and the Control system interact with the Weather stations. (what their goals are).



Each use case should be described using structured natural language

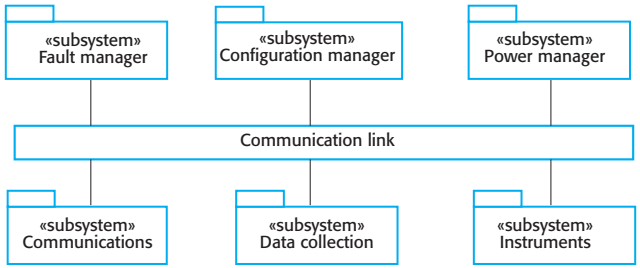
Use case description for "Report weather"

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

7.1.2 Architectural Design

- Start answering questions from Section 6.1 (use context and interactions to help answer them)
  - is there a generic application architecture?
  - is there an architectural pattern that might be used?
  - What will be the fundamental approach used to structure the system?
  - What strategy will be used to control the operation of the components in the system?
  - What about the non-functional requirements?
  - etc.
- Identify major components of the system and their interactions.

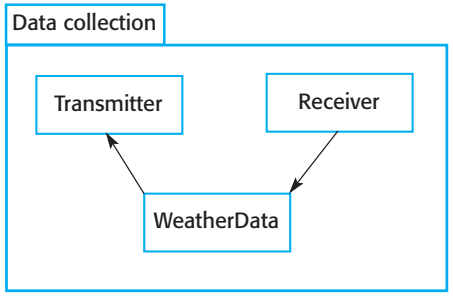
High level architecture of the weather station



New architectural pattern: (Message bus ?) Subsystems communicate by broadcasting messages on the communication link

Each subsystem listens for messages over the link and picks up messages intended for them.

Architecture of data collection system



Transmitter and Receiver manage communications.

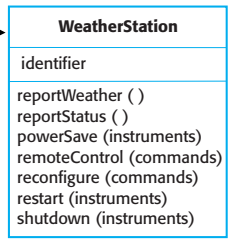
WeatherData object encapsulates the information collected from the instruments.

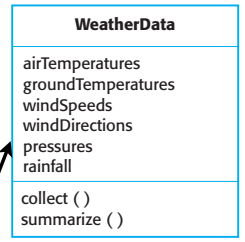
7.1.3 Object class identification

- Analyze descriptions of system (use cases, etc.):
  - nouns become objects and attributes
  - verbs become operations or services
- For the weather station, based on “report weather” use case:
  - objects representing the instruments that collect data
  - object representing the summary of weather data
  - high level object to encapsulate system interactions(s)
- Process is iterative and creative and cooperative
  - find objects, see how they should interact, this leads to realizing what other objects might be needed.

Weather station object classes

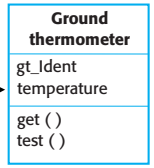
The basic interface of the weather station to its environment. Interactions from the use-case model.

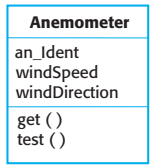


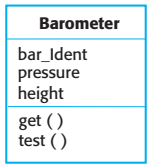


Encapsulates the summarized data from the instruments.

Application domain objects that are ‘hardware’ objects related to the instruments in the system.







Object class identification

- Focus on the objects, not the implementation
- Then refine the model
  - Look for common features and design the inheritance hierarchy (generalization)
  - Ex: Instrument superclass, with identifier, get(), test()
- The model should still be abstract, not "complete"

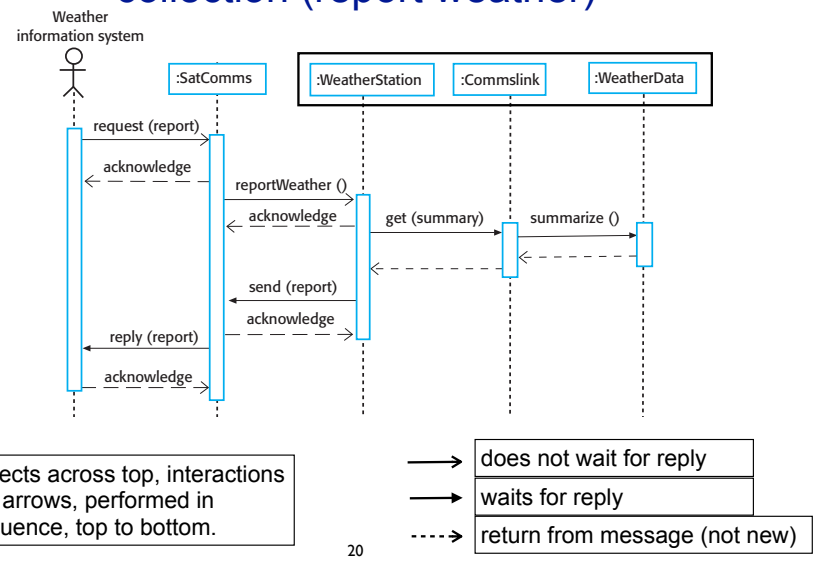
7.1.4 Design models (system models used for design)

- Design models show
  - the objects and object classes and
  - relationships and interactions between these entities.
- How much detail?
  - not much required if all system stakeholders are in close communication
  - much detail required when development is done by various teams, not in close contact with stakeholders.
- Which design models are needed?
  - depends on type of system, designer+implementor experience, development platform, software process, etc.

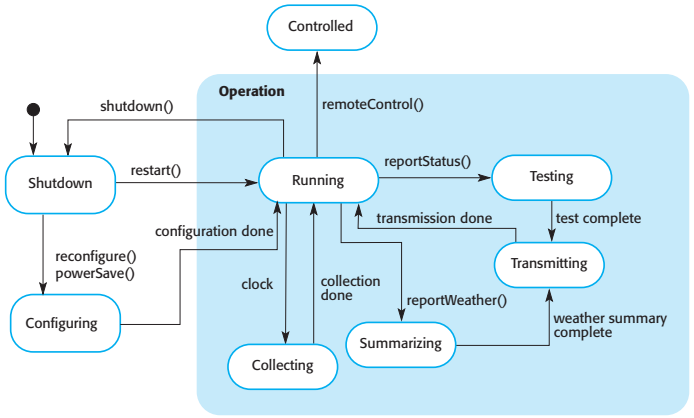
Design models: 2 types

- Structural models describe the static structure of the system in terms of object classes and relationships.
  - Class diagrams
  - Relationships: generalization, aggregation, uses/used-by
- Dynamic models describe the dynamic interactions between objects.
  - Sequence Diagrams (one per use case)
  - State Diagrams

Sequence diagram describing data collection (report weather)



Weather station state diagram



Only use state diagram if it is needed to clarify behavior

clock: stimulates instruments to take regular readings
other events are from use cases

7.1.5 Interface specifications

- General concept of interface
  - a point where two systems, subjects, organizations, etc., meet and interact.
- Software interface
  - the way one software component may interact with another
  - specific methods (functions) that may be called to access a software component
  - includes the signature of each function/method: names of method, types of each parameter.
  - can specify this using a class definition with no attributes, just methods

Interface specifications

- Why specify interfaces to components?
  - so components may be developed in parallel
  - one team develops component with the given interface.
  - another team develops component that accesses that component according to the interface.
- Interface is like a contract between components.
- Helps promote separation/independence.
  - can make changes behind the interface without affecting components using that interface.

7.2 Design patterns

- An design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.
- These solutions have been successful in previous projects (in various contexts).
- Patterns are a means of representing, sharing and reusing knowledge and experience.
- Pattern descriptions should include information about when they are and are not useful.
- Designer can browse pattern descriptions to identify potential candidates (see Design Patterns book).

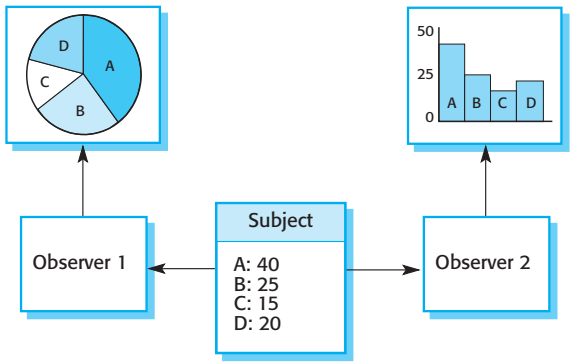
Design patterns: essential elements

- Name
  - A meaningful pattern identifier.
- Problem description
  - Explains the problem and its context.
  - Describes when the pattern (solution) may be applied.
- Solution description
  - A template for a design solution that can be instantiated in different ways. (Abstract, not concrete).
- Consequences
  - The results and trade-offs of applying the pattern.

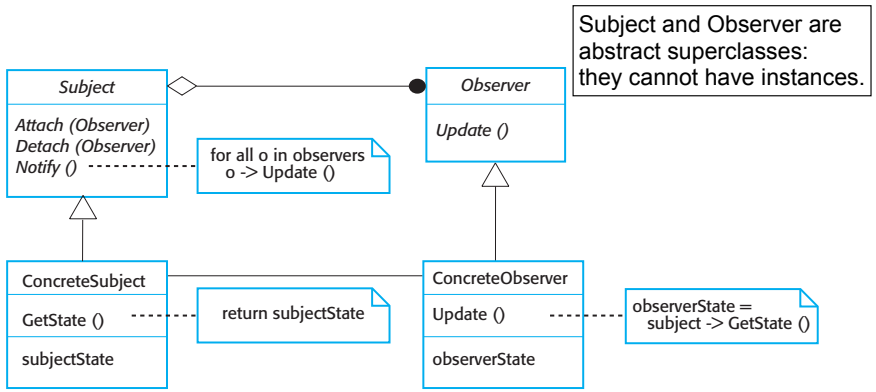
The observer pattern

Pattern name	Observer
Description	Defines a one-to-many dependency between objects so that when one object (the **subject**) changes state, all its dependents (the **observers**) are notified and updated automatically.
Problem description	In many situations, you need to maintain consistency between related objects. For example, using a GUI, you often have to provide multiple displays of state information, such as a graphical display and a tabular display. When the state is changed, all displays must be updated. This pattern may be used whenever it is not necessary for the object that maintains the state information to know about the objects that use the state information.
Solution description	This involves two main objects, Subject and Observer. The state is maintained in Subject, which has operations allowing it to add and remove Observers (the displays) and to issue a notification to the observers when the state has changed. The Observer maintains a (partial) copy of the state of Subject and implements the Update() method that is called by the Subject during notification of state changes. The Update() method asks the Subject for the updated state values that it needs.
Consequences	The Observer pattern lets you vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add and remove observers without modifying the subject or other observers. Since Update() provides no details on what part of the state changed, the observers may be forced to work hard to deduce the changes.

An instance of the observer pattern



A UML model of the Observer pattern



Other Design Patterns

- **Adapter**: Convert the interface of a class into another interface clients expect.
- **Façade**: Provide a unified interface to a set of interfaces in a subsystem.
- **Iterator**: Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented.
- **Abstract Factory**: Provide an interface for creating families of related or dependent objects without specifying their concrete classes
  - creating nodes of abstract syntax tree for different languages.

7.3 Implementation issues

- Aspects of implementation that are important to software engineering but not covered in programming textbooks
  - **Reuse**: developing software by reusing existing designs, components or systems
  - **Configuration management**: managing the different versions of each software component (the source code).
  - **Host-target development**: when the development (host) environment is on a different system from the production (target) environment.

7.3.1 Reuse: reuse levels

- The abstraction level
  - Don't reuse software directly but use knowledge of successful abstractions: Design/Architectural patterns.
- The object level
  - Directly reuse objects from a library rather than writing the code yourself.
- The component level
  - Components are collections of objects and classes that operate together to provide related functions (frameworks).
- The system level
  - Reusing entire application systems, requires configuration.

Reuse benefits+costs

- Benefits
  - Development should be quicker and cost less
  - Reused software should be reliable (well-tested).
- Costs
  - Time spent searching for and assessing candidates.
  - Expense of buying the reusable software.
  - Time spent adapting and configuring reusable software to fit your requirements
  - Time spent integrating various reusable components with each other and with new code.
- Always consider reusing existing knowledge and software when starting a new development project.

7.3.2 Configuration management

- Potential problems of team development
  - Interference: Changes made by one programmer could overwrite a change previously made by another.
  - Redo good work: Programmers accessing out-of-date versions could re-implement work already done.
  - Can't undo bad work: Figuring out how to undo problems introduced into a previously functioning system.
- Configuration management: Process of managing a changing software system, so all developers can
  - access code and documentation in a controlled way
  - find out what changes have been made
  - compile and link components to create the system.

Fundamental configuration management activities

- Version management
  - track different versions
  - coordinate work of multiple developers.
- System integration
  - define which versions of each component are used for a given version of the overall system.
  - then builds system automatically
- Problem tracking
  - allows users to report and track bugs.
  - allows developers to track progress on fixing bugs.

Configuration management tools

- Integrated tools: all three components in one
  - same interface, can link components together
  - ClearCase
- Version management
  - CMVC, CVS, subversion, git, mercurial.
- System integration (build tools)
  - make (unix), Apache Ant, or built into IDE
- Problem tracking
  - bugzilla
  - any database

7.3.3 Host-target development

- Host: computer on which software is developed
- Target: computer/system on which software runs
- Development platforms and execution platforms
  - hardware AND software (operating systems, databases, IDEs, configuration management, etc.)
- If the two platforms are not the same
  - deploy developed software to target for testing
  - test using a simulator on development machine
- If the two platforms ARE the same
  - developed software may still require supporting software not on development platform

Development platform tools

- Compiler(s)
- (Syntax-directed) editing system.
- Debugging system.
- Graphical editing tools: tools to edit UML models.
- Testing tools: Junit which can automatically run a set of unit tests on a new version of a program.
- Project support tools that help you organize the code for different development projects.
- Configuration management tools

7.4 Open source development

- The source code of the system is publicly available
- Volunteers are invited to participate in the development process (may be users).
- Some open source projects:
  - Linux, Apache web server, Java
  - Eclipse, FireFox, Thunderbird, Open Office
- Issues:
  - Should an open source approach be used for the software's development?
  - Should the system being developed (re)use open source software components?

Open source development

- How to make money developing open source products?
  - Development is cheaper: volunteer labor.
  - The company can sell support services
  - Software must have wide appeal
- Re-using open source software in software products:
  - These components are generally free.
  - These components are generally well-tested.
  - There may be licensing issues. . .

Open source licenses

- GNU General Public License (GPL).
  - reciprocal
  - if you re-use this open source software in your software then you must make your software open source.
- GNU Lesser General Public License (LGPL)
  - you can write components that link to open source code without having to publish the source of these components.
- Berkley Standard Distribution (BSD) License.
  - non-reciprocal
  - not obliged to re-publish any changes or modifications made to open source code.
  - you may include the code in proprietary systems that are sold.