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
  - i.e. Don't use UML (designed for object-oriented design) to write code in C.

Design and Implementation

- Software **Design**:
  - Creative activity, in which you:
  - 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
  - We assume you all have design and implementation experience.
- It is: to show how system modeling (ch 5) and architectural design (ch 6) are practiced in object oriented design, AND
- To introduce implementation issues not usually covered in programming books:
  - configuration management
  - open source development

7.1 Object-oriented design using UML

- Object-oriented system is 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.

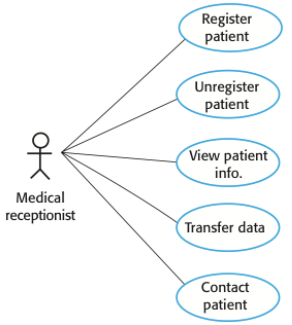
Object-oriented design and implementation activities

- During **requirements elicitation**, the client and developers define the purpose of the system.
- During **analysis**, developers aim to produce a model of the system that is correct, complete, consistent, and unambiguous.
- During **system design**, developers decompose the system into smaller subsystems that can be realized by individual teams.
- During **object design**, developers define solution domain objects to bridge the gap between the analysis model and the hardware/software platform defined during system design.
- During **implementation**, developers translate the solution domain model into source code.

1 Requirements elicitation

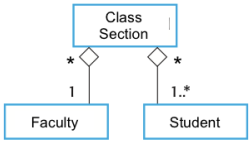
- Client and developers define the purpose of the system:
  - Develop use cases
  - Determine functional and non-functional requirements
- Major activities
  - Identifying actors.
  - Identifying scenarios.
  - Identifying use cases.
  - Refining use cases.

Use case diagrams



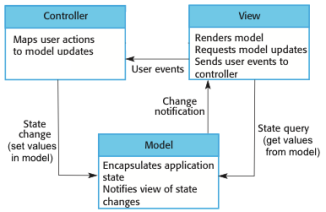
2 Object Oriented Analysis

- Developers aim to produce a model of the system that is correct, complete, consistent, and unambiguous
  - Model is a class diagram
  - Describing real world objects (only)
- Goal: transform use cases to objects
- Major activities
  - Identifying objects: entities from the real world
    - Look for nouns in use cases
  - Drawing sequence diagrams (external view)
    - Helps identify operations
  - Drawing the class diagram, with relationships
  - Drawing state diagrams as necessary

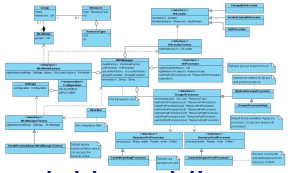


3 System Design (architecture)

- Developers decompose the system into smaller subsystems that can be realized by individual teams
- Goal:
  collect objects into subsystems
- Major activities
  - Identify major components of the system and their interactions (including interfaces).
    - Use architectural patterns
  - Identify design goals (non-functional requirements)
  - Refine the subsystem decomposition to address design goals



4 Object Design



- Developers complete the object model by adding implementation classes to the class diagram.
- Goal: bridge the gap between the analysis model and existing system components and resources.
- Major activities
  - Interface specification: define public interface of objects
  - Reuse:
    - frameworks, existing libraries (code)
    - design patterns (concepts) (see section 7.2)
  - Restructuring: maintainability, extensibility
  - Optimization: address performance goals

5 Implementation

- Developers translate the solution domain model (class diagram) into source code.
- Goal: map object model to code.
- Major activities
  - Map classes in model to classes in source language
  - Map associations in model to collections in source language
    - OO languages don't have "associations"
    - tricky: maintaining bidirectional associations
  - Refactoring
  - Persistence: storage of certain classes

```
#include <string>
#include <iomanip>
#include <sstream>
#include<iostream>
using namespace std;

// models a 12 hour clock
class Time        //new data type
{
private:
    int hour;
    int minute;
    void addHour();

public:
    void setHour(int);
    void setMinute(int);
    int getHour() const;
    int getMinute() const;

    string display() const;
    void addMinute();
};

// class function implementations

void Time::setHour(int hr) {
    hour = hr;          // hour is a member var
}
void Time::setMinute(int min) {
    minute = min;       // minute is a member var
}
int Time::getHour() const {
    return hour;
}
int Time::getMinute() const {
    return minute;
}

void Time::addHour() {  // a private member func
    if (hour == 12)
        hour = 1;
    else
        hour++;
```

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

- A 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).

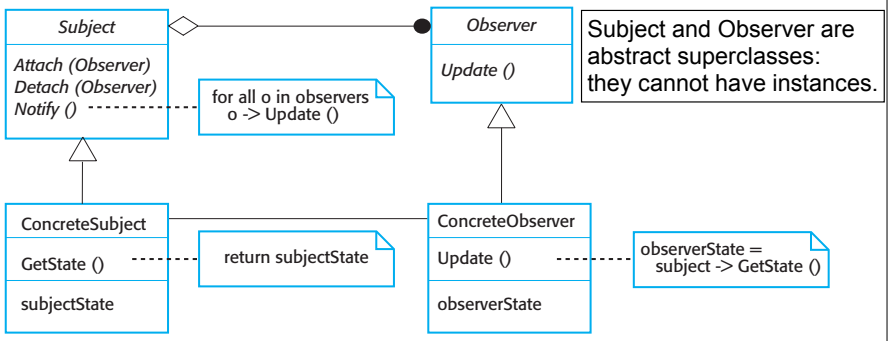
Design patterns: essential elements

- 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).
  - class diagram of cooperating classes
- Consequences
  - The results and trade-offs of applying the pattern.

The observer pattern

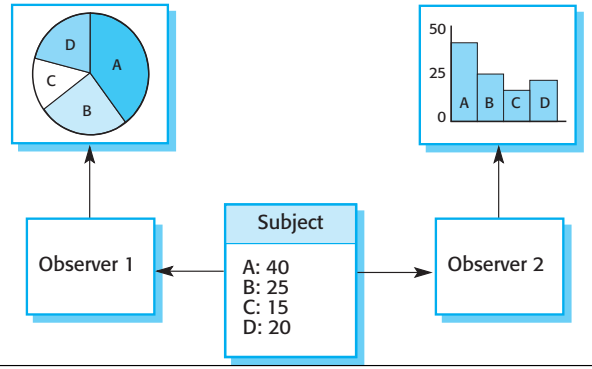
Pattern name	Observer
Description	Defines a one-to-many dependency between objects so that when one object (the **subject**) changes state, all its dependents (the **observers**) are notified and updated automatically.
Problem description	In many situations, you need to maintain consistency between related 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	See next slide (contains UML class diagram).
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.

A UML model of the Observer pattern



Observer involves two main objects, Subject and Observer. The state is maintained in Subject, which has operations allowing it to add and remove Observers 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.

An example using the observer pattern



The state is maintained in Subject.
Observer 1 and Observer 2 display different representations of the data.
Subject has a list containing Observer1 and Observer2, and it notifies them when the data changes (by calling Observer1.Update() and Observer2.Update()).
For each Observer, the Update() method asks the Subject for the updated state values that it needs, then redraws its image.

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. Example: JavaMail API
- The component level
  - Components are collections of objects and classes that operate together to provide related functions (frameworks).
  - Example: Java Swing (to implement GUIs)
- 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 of the files in the program
  - coordinate work of multiple developers.
- System integration
  - define which versions of each component and/or file 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
  - tools share same interface, can share information
  - 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: machine on which software is developed (development platform)
- Target: machine on which software runs (execution platform)
- Platforms include hardware AND software
  - operating systems, databases, development tools, etc.
- If the two platforms are not the same
  - deploy developed software to target for testing
  - or 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

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.