Lecture 3 : Object Oriented Design

(chapter 3. of Timothy Budd's "Intro to OOP" book)

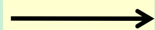
Acknowledgement : courtesy of Prof. Timothy Budd lecture slides

OOP Technique Example : Responsibility Driven Design

- OOP = a set of interacting objects
- Each object is a component that has responsibility in a community.
- System is divided into various activities that the system has to perform. Then, those activities(responsibilities) are distributed among objects in the system.
- After completing a list of responsibilities, we need to decide a set of objects and determine which responsibilities should be assigned to which object.
- Responsibilities in an object should be closely related.

Design Process

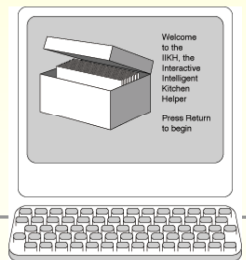
- Specification
  - Imprecise
  - Ambiguous
  - Unclear



- Software System Design
  - concrete logic

Development Process

1. Initial Description(Specification)
2. Refine Specification
3. Identification of Components
4. Identification of Components Responsibilities
5. Formalize the interface
6. Designing the subsystem
7. Implementing components
8. Integration of components
9. Maintenance & Evolution



An Example, the IIKH

IIKH(*Intelligent Interactive Kitchen Helper)*

- Imagine you are the chief software architect in a major computing firm
- The president of the firm rushes into your office with a specification for IIKH, the next PC-based product. It is drawn on the back of a dinner napkin

1.Initial Description

- Briefly, IIKH is a PC-based application
  - that will replace the box of index cards of recipes in the average kitchen. But more than simply maintaining a database of recipes, the kitchen helper assists in the planning of meals for an extended period, say a week. The user of the IIKH can sit down at a terminal, browse the database of recipes, and interactively create a series of menus. The IIKH will automatically scale the recipes to any number of servings and will print out menus for the entire week, for a particular day, or for a particular meal. And it will print an integrated grocery list of all the items needed for the recipes for the entire period.
- Initial description : Ambiguous, Unclear
- We need to clarify the ambigueties in the description.

2. Refine Specification

(Working Through Scenarios)

- Because of the ambiguity in the specification, the major tool we will use to uncover the desired behavior is to walk through application scenarios
  - Pretend we had already a working application. Walk through the various uses of the system
  - Establish the ``look and feel'' of the system
  - Make sure we have uncovered all the intended uses
  - Develop descriptive documentation
  - Create the high level software design
- Other authors use the term ``use-cases'' for this process of developing scenarios
  - 시스템 사용에 대한 시나리오의 집합

Scenario Example

Simple Browsing

Alice Smith sits down at her computer and starts the IIKH. When the program begins, it displays a graphical image of a recipe box, and identifies itself as the IIKH, product of IIKH incorporated. Alice presses the return button to begin.

In response to the key press, Alice is given a choice of a number of options. She elects to browse the recipe index, looking for a recipe for Salmon that she wishes to prepare for dinner the next day. She enters the keyword Salmon, and is shown in response a list of various recipes. She remembers seeing an interesting recipe that used dill-weed as a flavoring. She refines the search, entering the words Salmon and dill-weed. This narrows the search to two recipes.

She selects the first. This brings up a new window in which an attractive picture of the finished dish is displayed, along with the list of ingredients, preparation steps, and expected preparation time. After examining the recipe, Alice decides it is not the recipe she had in mind. She returns to the search result page, and selects the second alternative.

Examining this dish, Alice decides this is the one she had in mind. She requests a printing of the recipe, and the output is spooled to her printer. Alice selects “quit” from a program menu, and the application quits.

Abilities of the IIKH

- Here are some of the things a user can do with the IIKH
  - Browse a database of recipes
  - Add a new recipe to the database
  - Edit or annotate an existing recipe
  - Plan a meal consisting of several courses
  - Scale a recipe for some number of users
  - Plan a longer period, say a week
  - Generate a grocery list that includes all the items in all the menus for a period
  - …

3. Identification of Components

- Component
  - an abstract design entity with which we can associate responsibilities for different tasks
  - must have a small well defined set of responsibilities
  - should interact with other components to the minimal extent possible

4. Identification of Components Responsibilities

- CRC Cards
  - Components are most easily described using CRC cards

Component Name	Collaborators
Description of the responsibilities assigned to this component	*List of other components*

Greeter	Collaborators
Display Informative Initial Message Offer User Choice of Options Pass Control to either Recipe Database Manager Plan Manager for processing	Database Manager Plan Manager

The first component, The Greeter

- When the application is started, the Greeter puts an informative and friendly welcome window (the greeting) on the screen
- Actions
  - Casually browse the database of recipes
  - Add a new recipe
  - Edit or annotate a recipe
  - Review a plan for several meals
  - Create a plan of meals

The Recipe Database Component

- Actions
  - Must Maintain the Database of recipes
  - Must Allow the user to *browse* the database
  - Must permit the user to edit or annotate an existing recipe
  - Must permit the user to add a new recipe

Responsibilities of a Recipe

- Tasks
  - Maintains the list of ingredients and transformation algorithm
  - Must know how to edit these data values
  - Must know how to interactively display itself on the output device
  - Must know how to print itself
  - etc

The Planner Component

- Permits the user to select a sequence of dates for planning
- Permits the user to edit an existing plan
- Associates with *Date* object

The Date Component

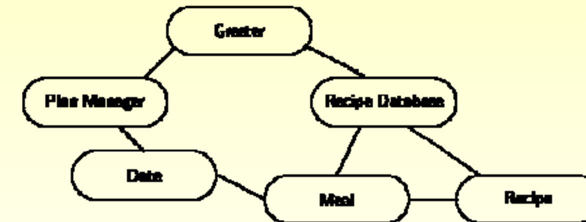
- User can edit specific meals
- User can annotate information about dates
  - ''Bob's Birthday'', ``Christmas Dinner'', …
- Can print out grocery list for entire set of meals

The Meal Component

- holds information about a single meal
  - Allows user to interact with the recipe database to select individual recipes for meals
  - User sets number of people to be present at meal, recipes are automatically scaled
  - Can produce grocery list for entire meal, by combining grocery lists from individual scaled recipes

The Six Components

- everything can be accomplished using only six software components



- You can at this point assign the different components to different programmers for development

Characteristics of Components

- Behavior and State
- Instances and Classes
- Coupling and Cohesion
- Interface and Implementation

Behavior and State

- Behavior
  - the set of actions a component can perform
- State
  - all the information (data values) held within a component
- it is common for behavior to change state

Instances and Classes

- There are likely many *instances* of recipe, but they will all *behave* in the same way
- We say the behavior is common to the class Recipe

Coupling and Cohesion

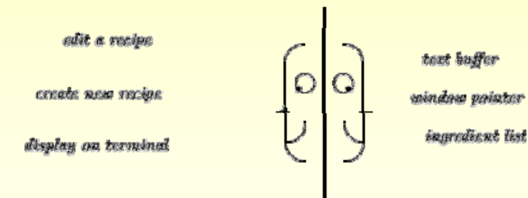
- Cohesion
  - the degree to which the tasks assigned to a component seem to form a meaningful unit.
  - Want to maximize cohesion
- Coupling
  - the degree to which the ability to fulfill a certain responsibility depends upon the actions of another component
  - Want to minimize coupling

Interface and Implementation

- We have characterized software components by what they can do
- The user of a software component need only know what it does, not how it does it

Two views of a Software System

- *information hiding*
  - purposeful hiding of implementation details



Public and Private View

- Public view
  - those features (data or behavior) that other components can see and use
- Private view
  - those features (data or behavior) that are used only within the component

5. Formalize the Interface

- formalize the channels of communication between the components
  - The general structure of each component is identified
  - Components with only one behavior may be made into functions
  - Components with many behaviors are probably more easily implemented as classes
  - Names are given to each of the responsibilities – these will eventually be mapped on to procedure names
  - Function parameters should be identified
  - Information is assigned to each component and accounted for
  - Scenarios are replayed in order to ensure all data is available

The selection of names

- should be evocative in the context of the problem
- should be short
- should be pronounceable (read them out load)
- Names should be consistent within the project
- Avoid digits within a name

6. Design

- Transform description into SW layout
- Design data structures in each component
- Transform behavior description into algorithms.

7. Implementing Components

- **implement** the desired activities in **each component**
- **Unit testing**: Subsystems are validated individually
  - Identify necessary conditions for correct functioning.
  - Test input values to verify that SW component will perform correctly when presented with legal input values

8. Integration of Components

- Components are slowly integrated into completed system, using stubs as yet unimplemented parts.
- Stubs
  - simple dummy routines with no behavior or with very limited behavior
- Errors discovered during integration cause reinvestigation of validation techniques performed at the subsystem level.

9. Maintenance and Evolution

- Software Maintenance
  - Activities subsequent to the delivery of the initial working version of SW system
- Software does not remain fixed after the first working version is released
  - Errors or bugs can be discovered. Must be corrected
  - Requirements may change. Say as a result of government regulations, or standardization among similar products
  - Hardware may change
  - Users expectations may change. Greater functionality, more features. Often as a result of competition from similar products
  - Better documentation may be required
- Good Design
  - Recognizes the inevitability of changes and plans an accommodation for them from the very beginning

Prepare for Change!

- User requirements change with experience, hardware changes, government regulations change.
  - Try to predict the most likely sources of change, and isolate the effect. Common changes include interfaces, file formats, communication protocols
  - Isolate interfaces to hardware that is likely to change
  - Reduce dependency of one software component on another
  - Keep accurate record of the reasoning behind every major decision in the design documentation.

Common Design Flaws

- Direct modification
  - Components that make direct modification of data values in other components are a direct violation of encapsulation
- Too Much Responsibility
  - Components with too much responsibility are difficult to understand and to use
- No Responsibility
  - Components with no responsibility serve no purpose
- Components with unused responsibility
  - Usually the result of designing software components without thinking about how they will be used
- Misleading Names
  - Names should be short and unambiguously indicate what the responsibilities of the component involve