Design

What do we mean by program *design*?

Why do we worry about design when writing a program? Why isn't it enough that the program works?

Suppose the code is never intended to be read by anyone else, or used again?

Outline for Week 7

I. Design criteria

II. The CRC-card method

   Flight reservation
   Address book
   Course registration

III. Invitation class

O-o design: The CRC-card method

In writing object-oriented software, it is very important to get the design right.

If the design is wrong,

- Objects of one class may need to make extensive use of features of another class ("high coupling").

  It's OK if objects of one class merely *use* public features of another class. But if you find your code depending on the implementation of the other class, your code becomes unmaintainable.

- Methods and instance variables grouped in one class have little relationship with each other (low cohesion).

To get the design right, we should be careful to choose our classes.

The goals of this process are to—

- Discover classes.
- Determine the responsibilities of each class.
- Describe the relationships among the classes.

To discover the classes, we can look for the *nouns* in the task description (sometimes called the “requirements document”).

For example, if I say,

> The function of the system is to allow bus riders to plan a route from origin to destination,

what might be the classes?

When choosing classes, make sure that what you identify …

- is a singular noun,
- does not really have the same functionality as some other class,
- is not simply a primitive type or a library object,

Now let’s consider a sample system.

Example 1. Flight reservation

Requirements for the Flight Reservation System

- The mission is to allow round-trip airline *tickets* to be bought over the Web.
- Each *customer* specifies an origination *airport*, a destination airport, and dates for outbound and return *flights*.
- The customer reserves one *outbound flight* and one *return flight* from a menu presented by the system.
- Each choice that the system presents consists of one or more flight *segments* (there may be a stop or a change of planes).
- The customer may buy tickets for one or more *passengers*.
- No more tickets can be sold for a flight than there are *seats* on the *plane*.
- Each passenger is assigned to a specific seat.
- The system calculates the total *cost* of the tickets by adding the cost of the individual segments.
- If dissatisfied with the cost, the customer may select *alternate flights.*

- After a customer has bought a ticket, (s)he will be e-mailed a *confirmation*

Take a couple of minutes working with your group to identify the classes. Then enter your class names here.

Also name some nouns that are not classes.

(*Note:* Be sure to avoid this common misconception: Something that is an *attribute* of another class may be a class itself!)

Example 2. Address book

Here is a very complete example of an address book.

We will work our way from the requirements statement, through use cases to CRC cards.

Responsibilities and collaborators

Finding the classes is only the first step in the design process.

Next, we need to look for *responsibilities*, which are usually *verbs* in the task description.

For each responsibility, there may be one or more *collaborators*—classes that need to be called to help fulfill the responsibility.

In summary, we have—

- *Classes:* To find the objects, look for the nouns.
- *Responsibilities:* Things a class knows or can do.
- *Collaborators:* Other classes that are *directly* involved in fulfilling these responsibilities.

Now let's consider some responsibilities of the Customer class in the Flight Reservation system. Which collaborator(s) does each one have? Enter responsibilities and collaborators here.

CRC cards

A common design practice is to write information for each class on a separate card. A card has the form …

Class Name	
Responsibility 1	Collaborator(s) 1
Responsibility 2	Collaborator(s) 2
…	…
Responsibility *n*	Collaborator(s) *n*

We don't have a good way for you to share entire CRC cards with the rest of the class, but you can simulate a CRC card by filling out this class/responsibility/collaborator form repeatedly.

Common errors in CRC-card design

In designs created by students, certain errors keep coming up over and over. Here are some examples.

1. *Using a class name that is not a singular noun.*

   "Customers", "Segments", "Buy"

2. *Naming a system class as a key abstraction of the program.*

   "String", "Date"

3. *Defining a new class where an existing (usually primitive) object would suffice.*

   "Cost", "Time"

4. *Thinking that something can't be a key abstraction because it is part of a larger abstraction.*

   "Seat" can't be a key abstraction, because it's part of the plane.

   "Wheel" can't be a key abstraction, because it's an attribute of the plane.

5. *Treating collaboration as a transitive relationship.*

   Class: Customer
   Responsibility: Buy ticket

Collaborators: Passenger, Flight, Segment, Airport

Let's see if we can find any of these errors in your designs for the Flight Reservation System.

Example 3. Invitations

Consider inviting someone to join your team in Expertiza. What should be the responsibilities of an Invitation class?

What does the code need to do when someone accepts an invitation?

Now let's see how the current Invitation class is implemented in Expertiza.

Does it give us any hints on what else needs to be done when an invitation is accepted?

What else does the code need to do when someone accepts an invitation?

Now let's design the **`ProjectTopic`** class. What are the responsibilities of **`ProjectTopic`**?

Now let's look at the current SignupTopic class.

Do we need an **`invitations_controller`**?