Designing Applications

Chapter-13;
Objects First with Java using BlueJ

Analysis and Design

- Until now, we have described how to write good classes.
- We have assumed that we (more or less) know what the classes are.
- In real software projects, deciding what classes to implement is often the most difficult tasks.
- This is generally referred to as the *analysis and design* phase.

Analysis and Design

- A large and complex area.
- The *verb/noun method* is a fairly simple method, suitable for relatively small problems.
- *CRC cards* support the design process.

The verb/noun Method

- In a project definition/description:
- The nouns refer to 'things'.
  - A source of classes and objects.
- The verbs refer to 'actions'.
  - A source of interactions between objects.
  - Actions are behavior, and hence methods.

Example Project Description

- The cinema booking system should store seat bookings for multiple theaters.
- Each theater has seats arranged in rows.
- Customers can reserve seats and are given a row number and seat number.
- They may request bookings of several adjoining seats.
- Each booking is for a particular show.
  (i.e., the screening of a given movie at a certain time)
- Shows are at an assigned date and time, and scheduled in a theater where they are screened.
- The system stores the customer's phone number.

Nouns and Verbs

Cinema booking system
Stores (seat bookings)
Stores (phone number)

Theater
Has (seats)

Movie

Customer
Reserves (seats)
Is given (row number, seat number)
Requests (seat booking)

Time

Date

Seat booking

Show
Is scheduled (in theater)

Seat

Seat number

Telephone number

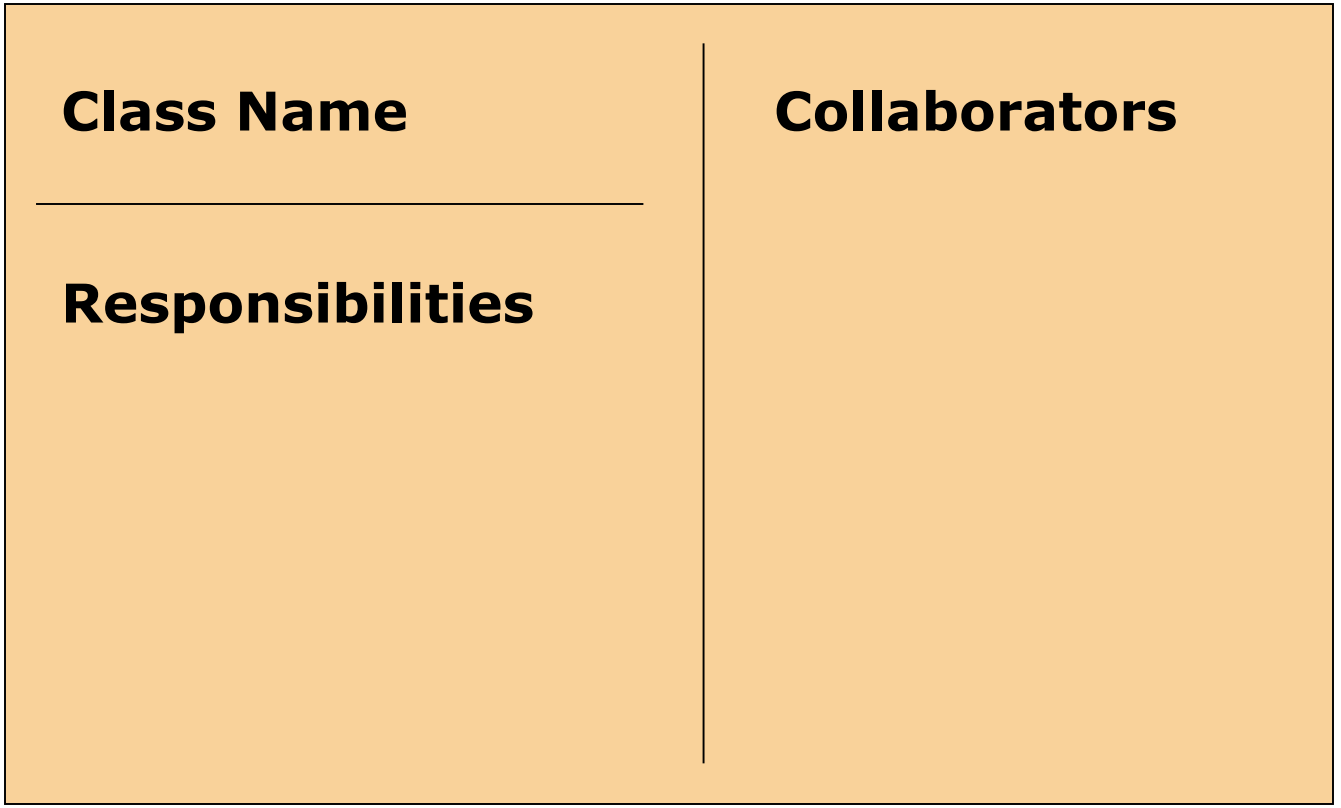
Row

Row number

Using CRC cards

- CRC stands for Class/Responsibilities/Collaborators.
- First described by Kent Beck and Ward Cunningham, in 1989.
- Each index card records:
  - A *class* name.
  - The class's *responsibilities*.
  - The class's *collaborators*.

A CRC card



Scenarios

- An activity that the system has to carry out or support.
  - Sometimes known as *use cases*.
- Used to discover and record object interactions (collaborations).
- Can be performed as a group activity.

Scenarios

A customer visits the booking system and wants to make a reservation for two seats tonight to watch the classic movie "The Shawshank Redemption".

Making a reservation involves finding and reserving seats.

Example

CinemaBookingSystem	**Collaborators**
• Can find shows by title and day. • Stores collection of shows. • Retrieves and displays show details. • Retrieves and displays theater seats. • Reserves seat numbers of a show. • ...	**Show** **Collection** **Seat** **Theater** **Customer**

Scenarios as Analysis

- Scenarios serve to check the problem description is clear and complete.
- Sufficient time should be taken over the analysis.
- The analysis will lead into design.
  - Spotting errors or omissions here will save considerable wasted effort later.

Class design

- Scenario analysis helps to clarify application structure.
  - Each card maps to a class.
  - Collaborations reveal class cooperation/object interaction.
- Responsibilities reveal public methods.
  - And sometimes fields; e.g. “Stores collection ...”

Designing Class Interfaces

- Replay the scenarios in terms of method calls, parameters and return values.
- Note down the resulting signatures.
- Create outline classes with public-method stubs.
- Careful design is a key to successful implementation.

Using Design Patterns

- Inter-class relationships are important, and can be complex.
- Some relationships recur in different applications.
- Design patterns help clarify relationships, and promote reuse.

Singleton

- Ensures only a single instance of a class exists.
  - All clients use the same object.
- Constructor is private to prevent external instantiation.
- Single instance obtained via a static **`getInstance`** method.
- Example: **`java.lang.Runtime`**

Singleton

```
public class Singleton {

   private static Singleton INSTANCE = null;

   private Singleton() {
      ...
   }

   public static Singleton getInstance() {
      if (INSTANCE == null)
         INSTANCE = new Singleton();
      return INSTANCE;
   }

   // Other public methods follow here
}
```

Decorator

- Augments the functionality of an object.
- Decorator object wraps another object.
  - The Decorator has a similar interface.
  - Calls are relayed to the wrapped object ...
  - but the Decorator can interpolate additional actions.
- Example: **`java.io.BufferedReader`**
  - Wraps and augments an unbuffered **`Reader`** object.

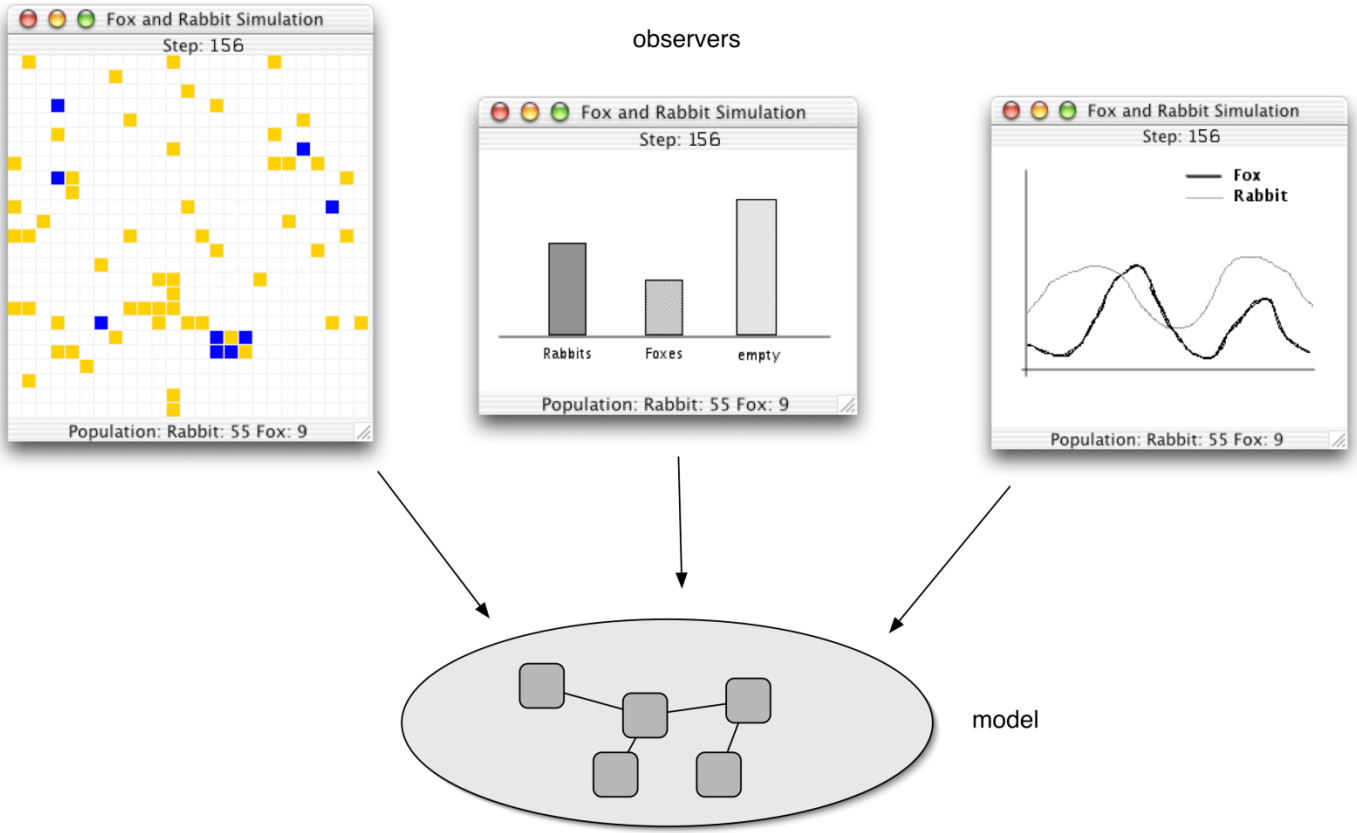
Factory Method

- Clients require an object of a particular interface type or superclass type.
- A factory method is free to return an implementing-class object or subclass object.
- Exact type returned depends on context.
- Example: **`iterator`** methods of **`java.util.Collection`** classes.

Observer

- Supports separation of internal model from a view of that model.
- Observer defines a one-to-many relationship between objects.
- The object-observed notifies all Observers of any state change.
- Example **`SimulatorView`** in the *foxes-and-rabbits* project.
- Examples: **`java.util.EventListener;`** **`java.util.Observer + java.util.Observable`**

Observers


Fox and Rabbit Simulation
Step: 156
Population: Rabbit: 55 Fox: 9
observers
Fox and Rabbit Simulation
Step: 156
Rabbits
Foxes
empty
Population: Rabbit: 55 Fox: 9
Fox and Rabbit Simulation
Step: 156
Fox
Rabbit
Population: Rabbit: 55 Fox: 9
model

Review

- Class collaborations and object interactions must be identified.
  - CRC analysis supports this.
- An iterative approach to design, analysis and implementation can be beneficial.
  - Regard software systems as entities that will grow and evolve over time.

Review

- Work in a way that facilitates collaboration with others.
- Design flexible, extendible class structures.
  - Being aware of existing design patterns will help you to do this.
- Continue to learn from your own and others' experiences.