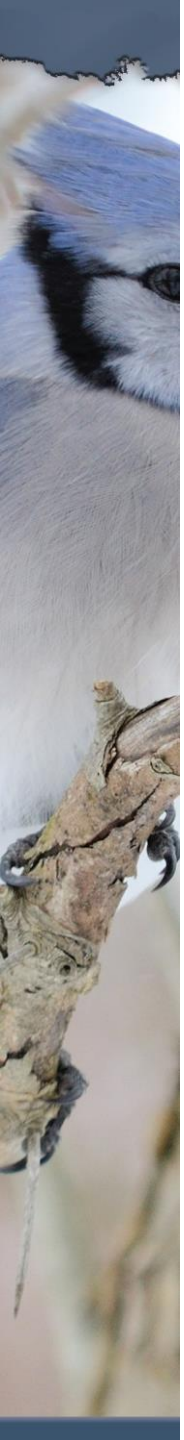
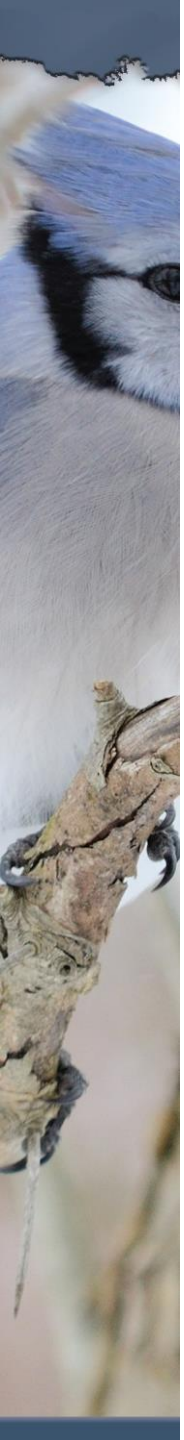


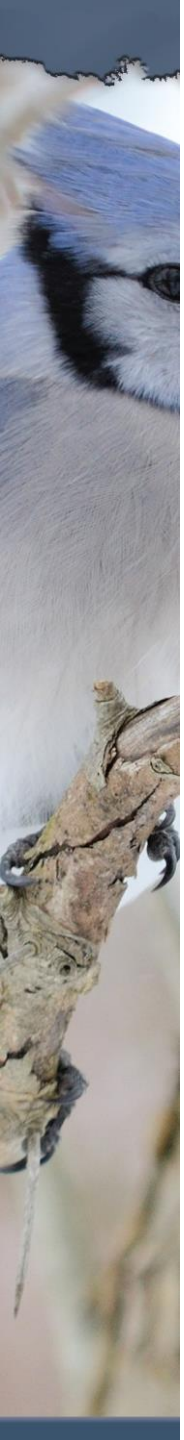
Designing applications

Edited by Ehsan Edalat and Amir Kalbasi



Software life cycle

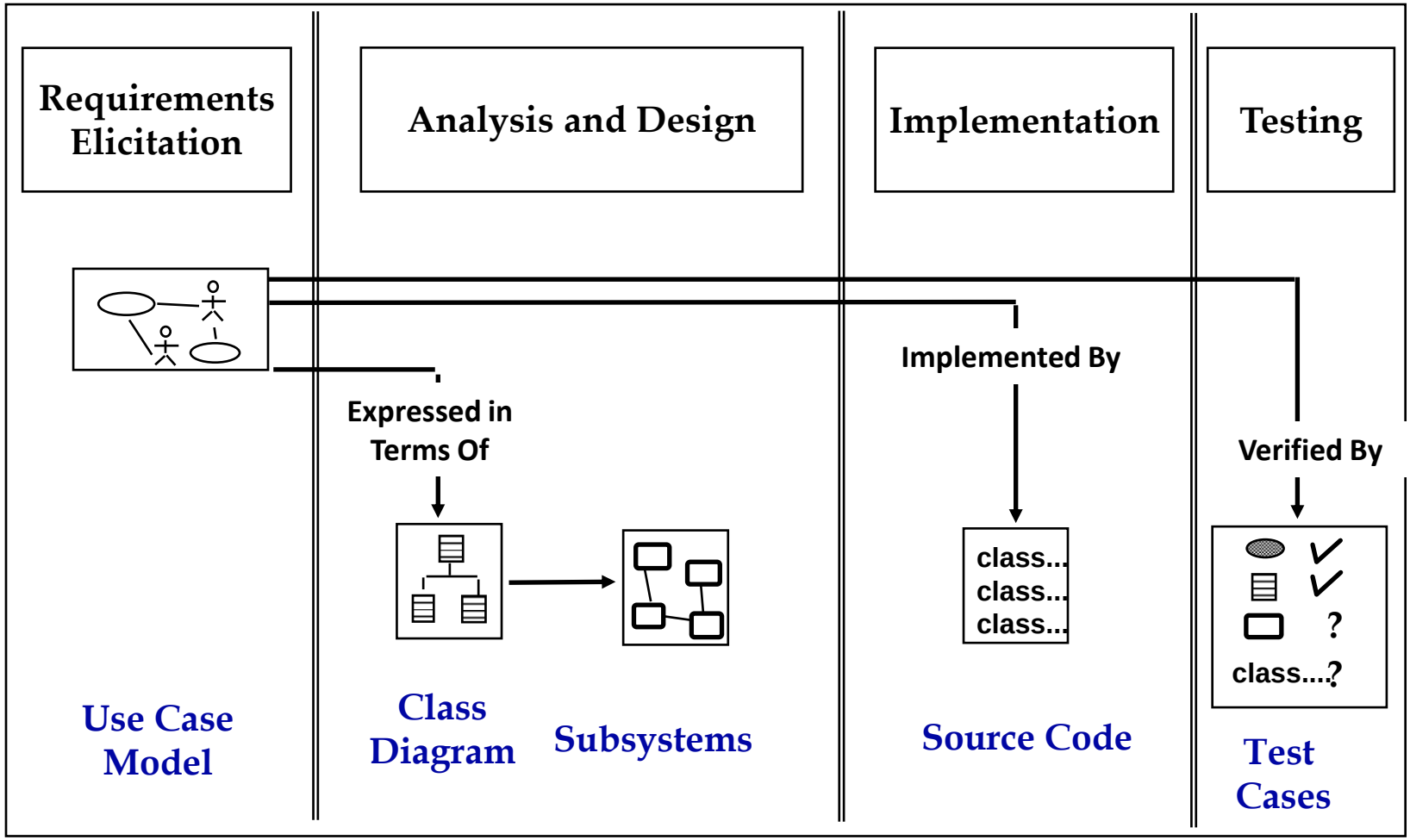
- Waterfall model.
  - Analysis
  - Design
  - Implementation
  - Unit testing
  - Integration testing
  - Delivery
- No provision for iteration.

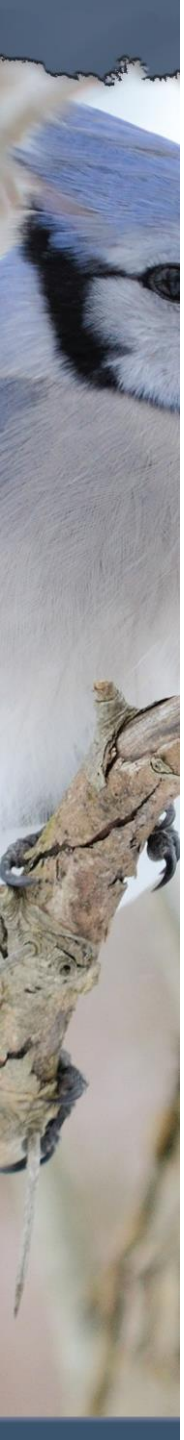


Iterative development

- Use early prototyping.
- Frequent client interaction.
- Iteration over:
  - Analysis
  - Design
  - Prototype
  - Client feedback
- A growth model is the most realistic.

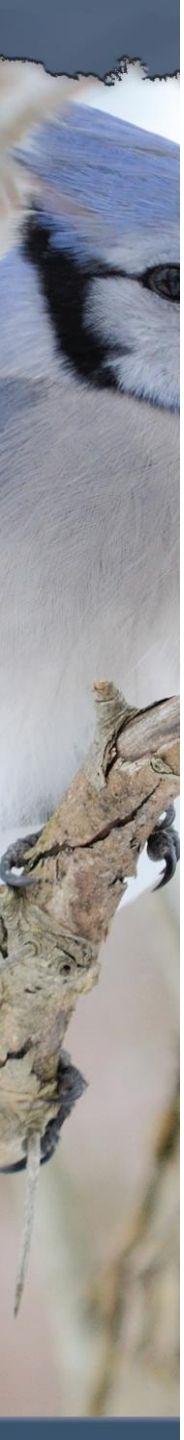
Software Lifecycle Activities





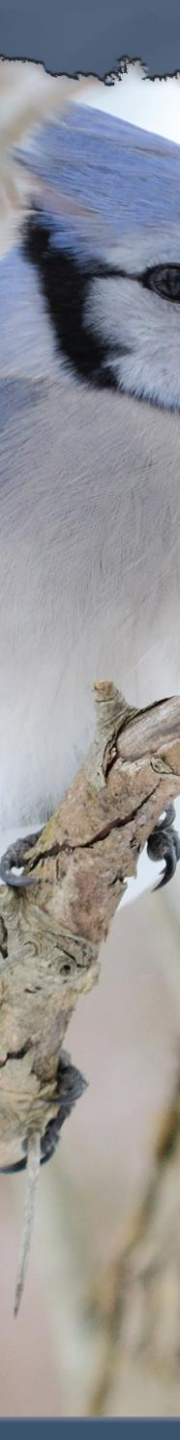
Scenarios

- 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 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.



Example scenario

Actor: **Bank Customer**

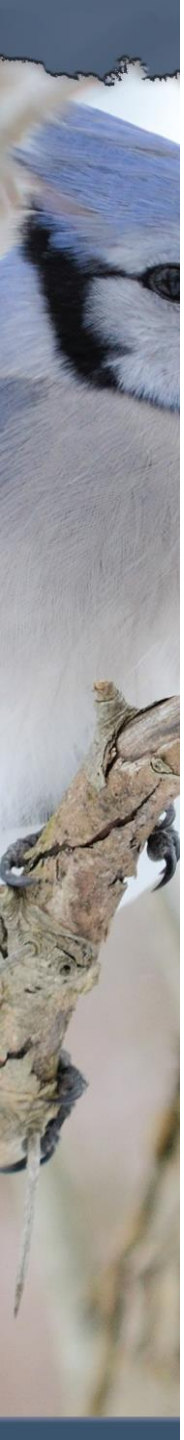
- Person who owns one or more Accounts in the Bank.

Withdraw Money

- Bank Customer specifies an Account and provides credentials to Bank proving that s/he is authorized to access Bank Account.
- Bank Customer specifies amount of money s/he wishes to withdraw.
- Bank checks if amount is consistent with rules of Bank and state of Bank Customer's account. If that is the case, Bank Customer receives money in cash.

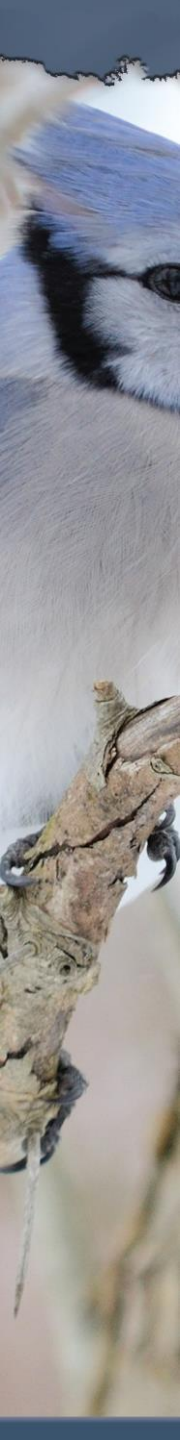
Scenario - flow of events

Actor steps	System steps
1. The Bank Customer inputs the card into the ATM.	
	2. The ATM requests a four-digit PIN.
3. The Bank Customer types in PIN.	
	4. PIN is checked. If several accounts are on the card, the ATM offers them for selection by the Bank Customer.
5. The Bank Customer selects an account.	
	6. The ATM requests the amount to be withdrawn by the Bank Customer.
7. The Bank Customer inputs an amount.	
	8. The ATM outputs the money, prints a receipt and ends the session.



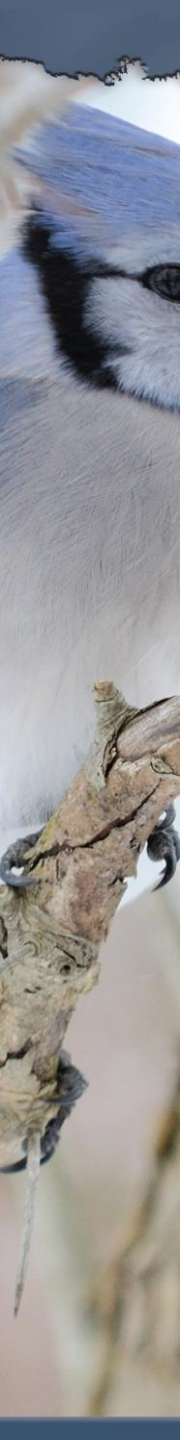
Identifying objects (design)

- Pick a Scenario and look at flow of events
- Do a textual analysis (noun-verb analysis)
  - Nouns are candidates for objects/classes
  - Verbs are candidates for methods
  - This is also called Abbott's Technique



A problem description (1)

- 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.



A problem description (2)

- 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)

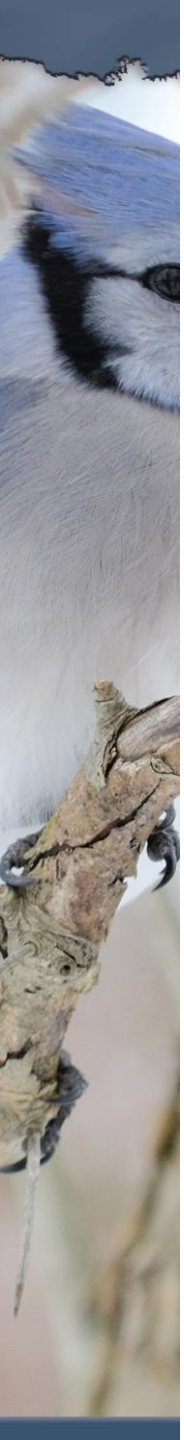
Seat

Seat number

Telephone number

Row

Row number



Using CRC cards

- First described by Kent Beck and Ward Cunningham.
- Each index card records:
  - A *class* name.
  - The class's *responsibilities*.
  - The class's *collaborators*.

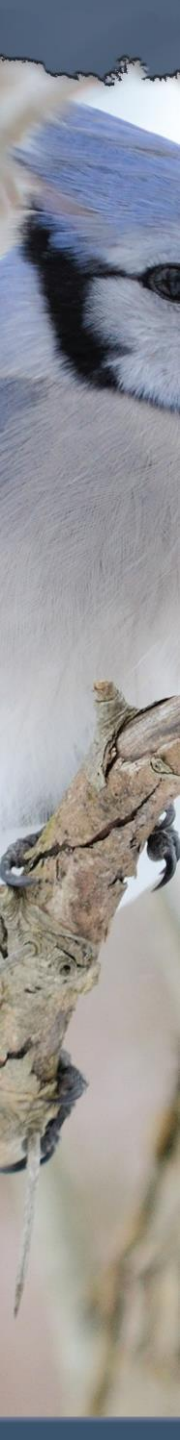
A CRC card

Class name	**Collaborators**
Responsibilities	

A partial example

CinemaBookingSystem	Collaborators

Can find shows by title and day.
Stores collection of shows.
Retrieves and displays show details.
...
	`Show`
	
	`Collection`

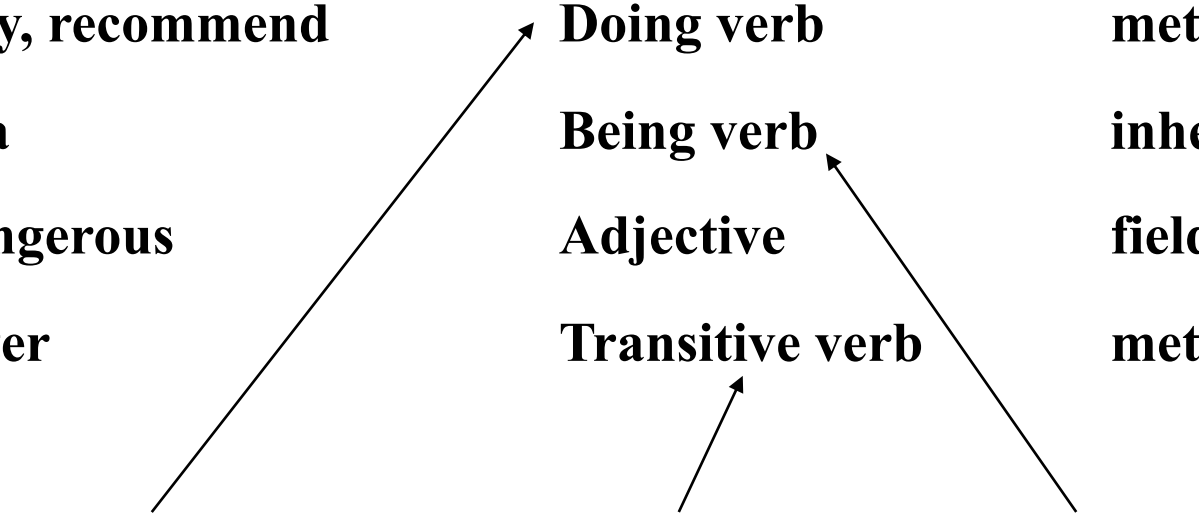


Class design

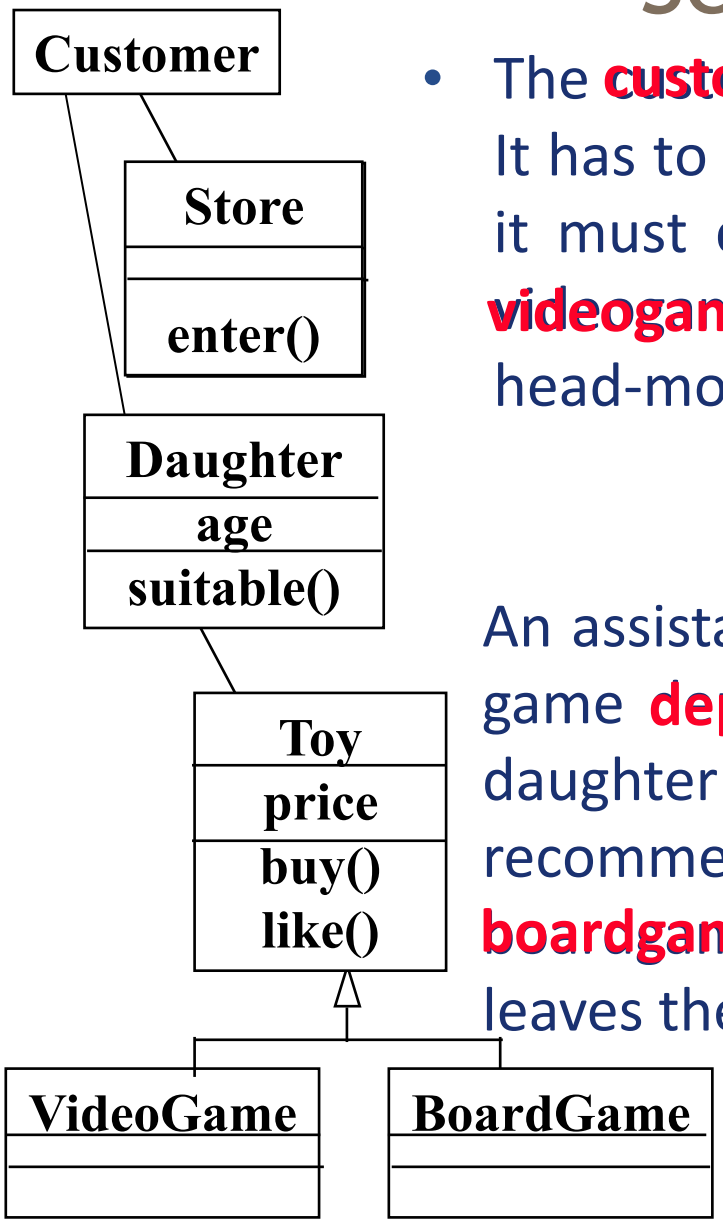
- Scenario analysis helps to clarify application structure.
  - Each card maps to a class.
  - Collaborations reveal class cooperation/object interaction.
- Responsibilities reveal public methods.
  - And sometimes fields; e.g., “Stores collection ...”

Mapping parts of speech to model components (Abbott's Technique)

Example	*Part of speech*	*Model component*
"Monopoly"	**Proper noun**	**object**
Toy	**Common noun**	**class**
Buy, recommend	**Doing verb**	**method**
is-a	**Being verb**	**inheritance**
dangerous	**Adjective**	**field**
enter	**Transitive verb**	**method**

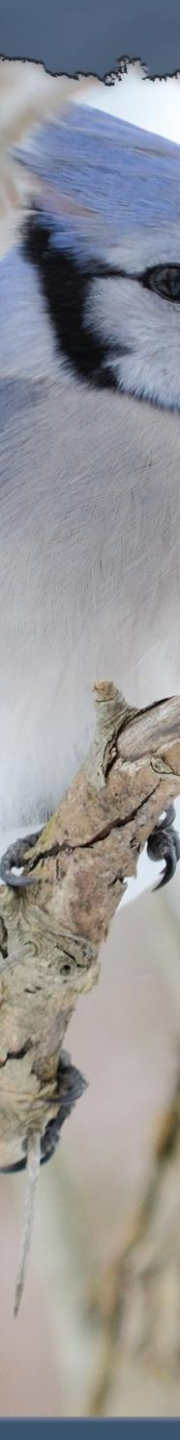


Generating a Class Diagram from Scenario



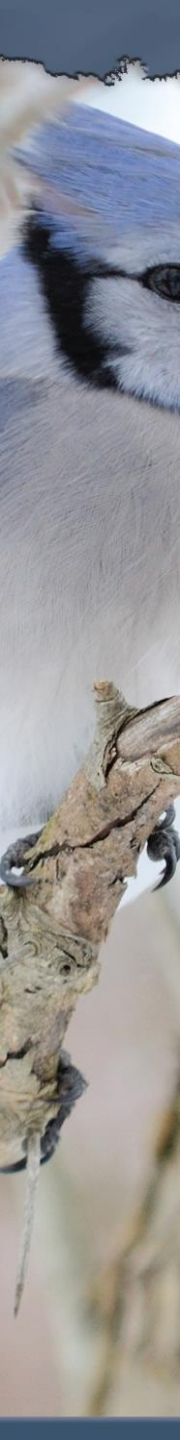
- The **customer enters** the **store** to **buy** a **toy**. It has to be a toy that his **daughter** likes and it must cost less than **50 Euro**. He tries a **videogame**, which uses a data glove and a head-mounted display. He likes it.

An assistant helps him. The suitability of the game **depends** on the **age** of the child. His daughter is only 3 years old. The assistant recommends another **type of toy**, namely a **boardgame**. The customer buy the game and leaves the store



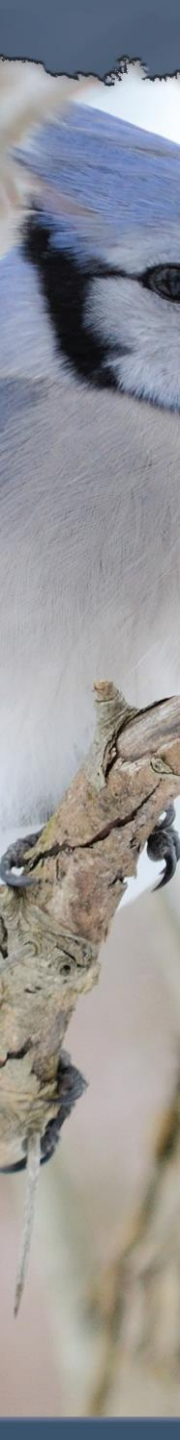
Using design patterns

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



Design Pattern

- A **design pattern** describes a common problem that occurs regularly in software development and then describes a general solution to that problem that can be used in many different contexts.
- For software design patterns, the solution is typically a description of a small set of classes and their interactions.



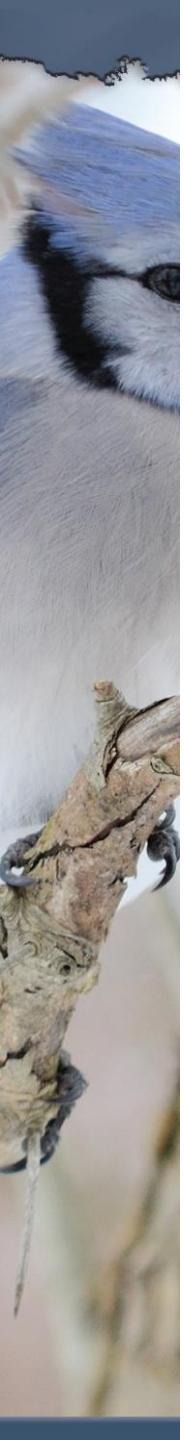
Singleton

- 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.
- Enums support the Singleton pattern.

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
}
```

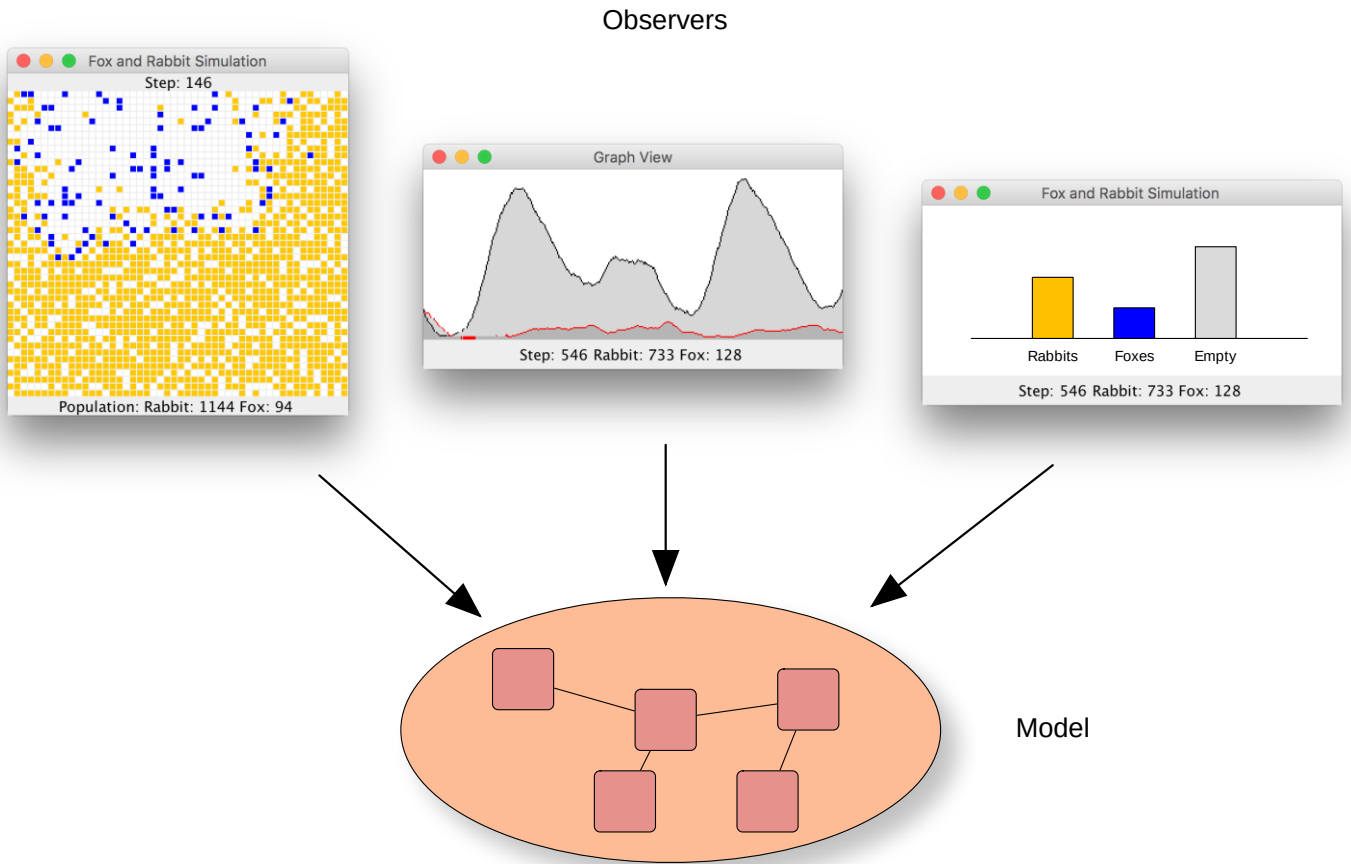
Think about a problem that there are at most 4 objects in the project. How can we implement it?



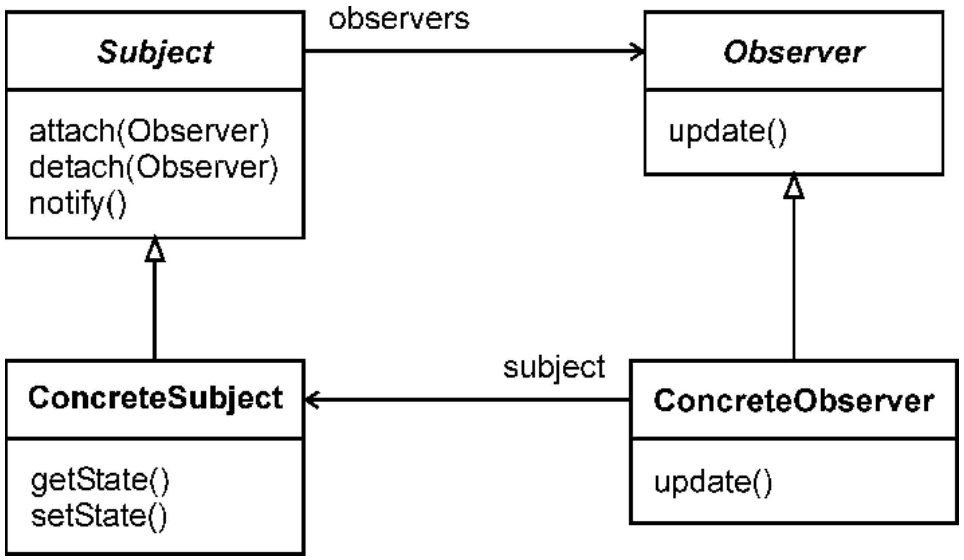
Observer

- 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*.

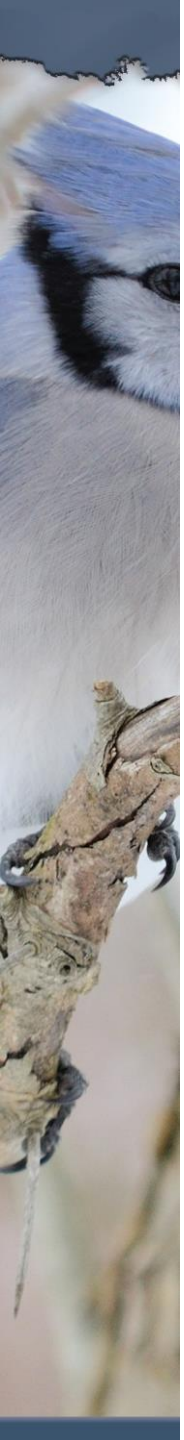
Observers



Observer Design Pattern

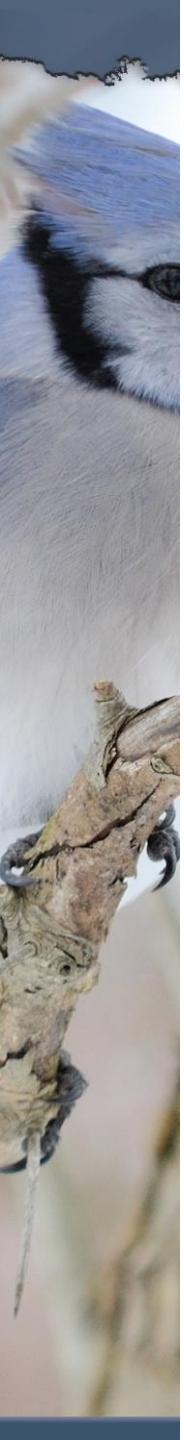


- https://www.tutorialspoint.com/design_pattern/observer_pattern.htm
- https://www.javaworld.com/article/2077258/observer-and-observable.html



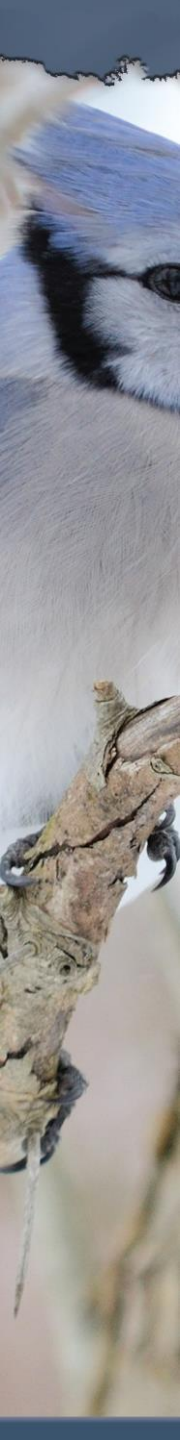
Observer Design Pattern

- Think about two robots, which imitate each other actions. How can we code these two with the Observer design pattern? What are the challenges of programming it?
- Code a simple Java program for practicing the question.



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.