# Software Architecture and Design

Terminology, finding classes, class responsibilities and collaborators, MVC pattern, best practices

# Terminology

- Software Engineering
  - Dictionary.com: “A systematic approach to the analysis, design, implementation and maintenance of software.”
- Software architecture
  - CMU SEI: “Software architecture serves as the blueprint for both the system and the project developing it, defining the work assignments that must be carried out by design and implementation teams. The architecture is the primary carrier of system qualities such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision. Architecture is an artifact for early analysis to make sure that a design approach will yield an acceptable system.”
- Software architect
  - Wikipedia: “Programmer who makes high-level design choices...”

# Introducing a problem

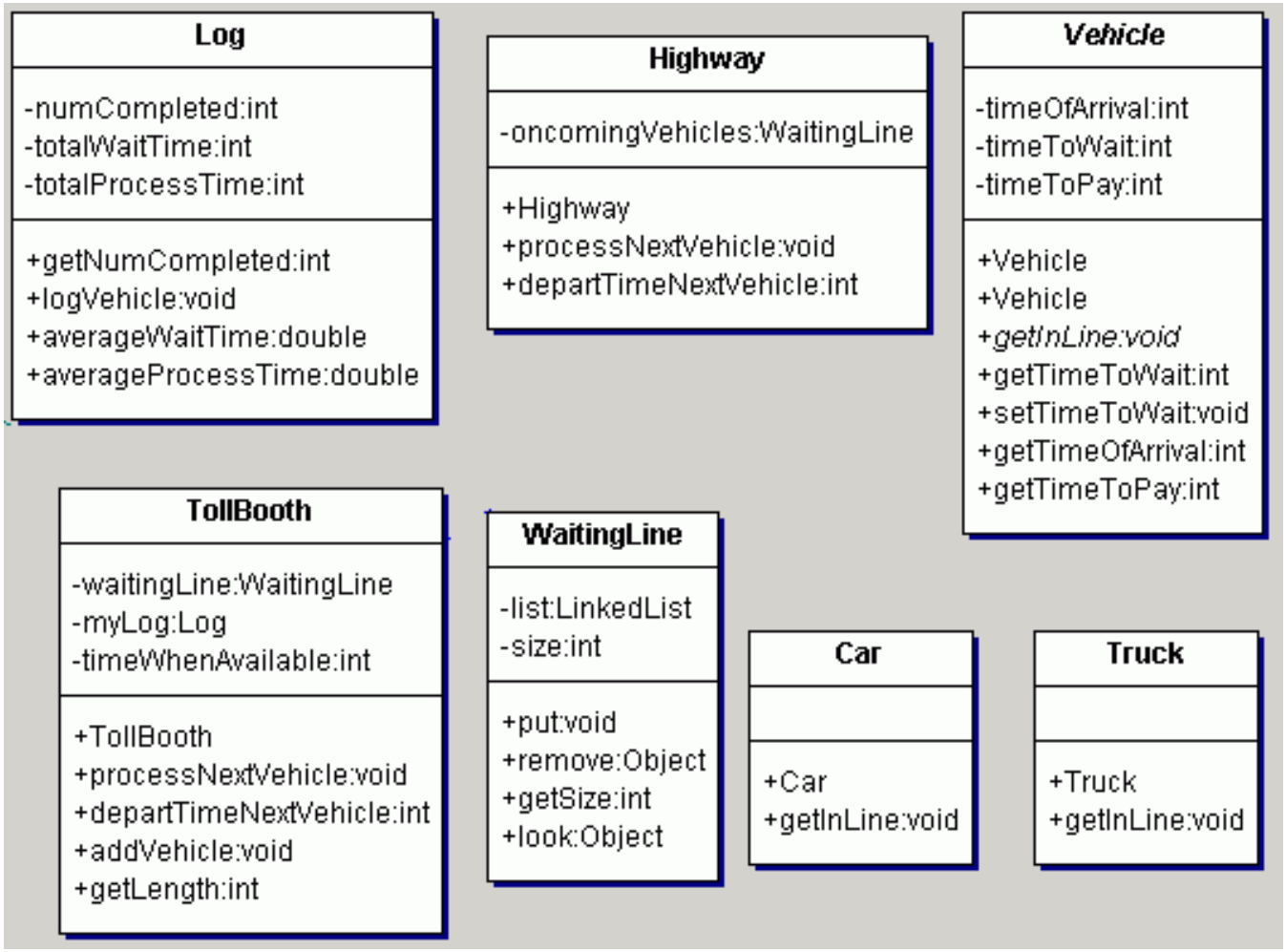
"In order to plan a toll highway, the highway department must estimate the wait times for cars and trucks that line up at toll booths. The highway department wants to simulate the activities around vehicles, the highway, and the toll booths. The simulation will keep track of the time each vehicle spends in line to pay its toll and how much it pays, logging this information as the vehicle leaves the toll booth line. The simulation log contains the running statistical results as they accumulate during the simulation."

# Abstracting the problem domain

- Look for nouns in a problem statement that tell the objects/classes that are important for modeling the **problem domain** (area undergoing analysis).
  - Not all nouns will translate into classes.
  - There will be some classes in the problem domain that aren't directly in the statement.

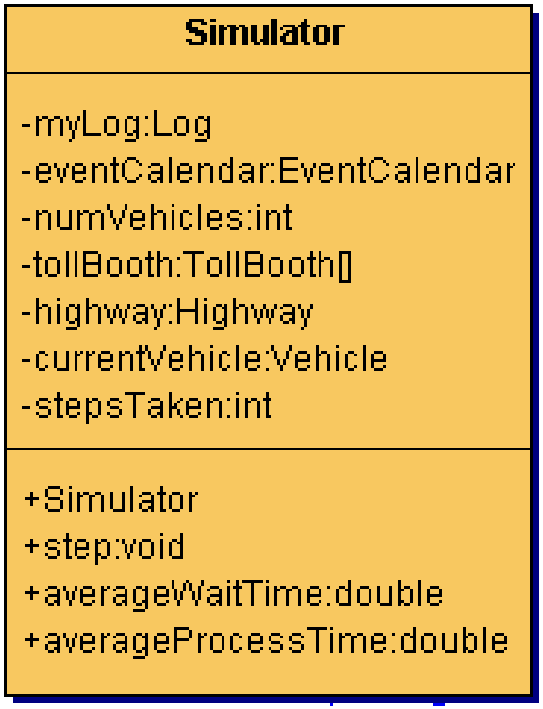
- Here are some nouns from the toll booth problem domain. Some are important in the model.
  - Highway Department
  - Vehicle - Car - Truck
  - Highway
  - TollBooth
  - WaitingLine
  - Log (to keep track of information)

# Capturing nouns that matter



# Don’t forget the obvious

The problem is a simulation. How about a simulator? – another model class.



# Look for classes outside the problem domain

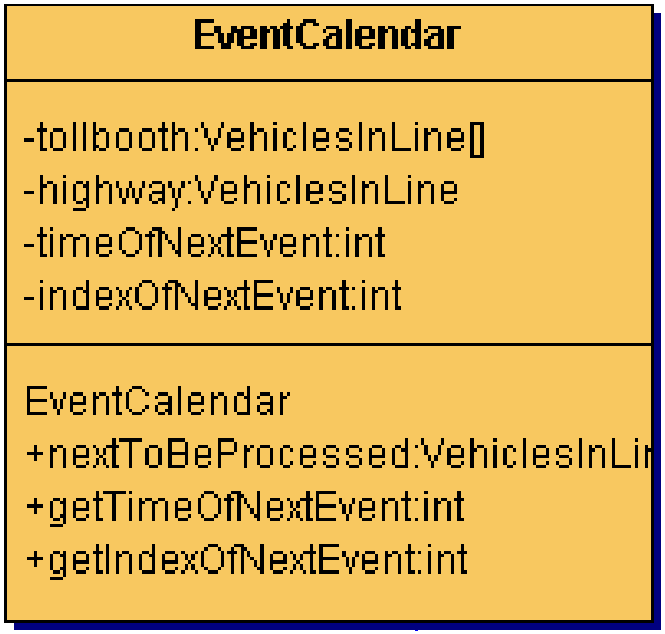
How do we get other classes? A suggestion from Timothy Budd – look for:

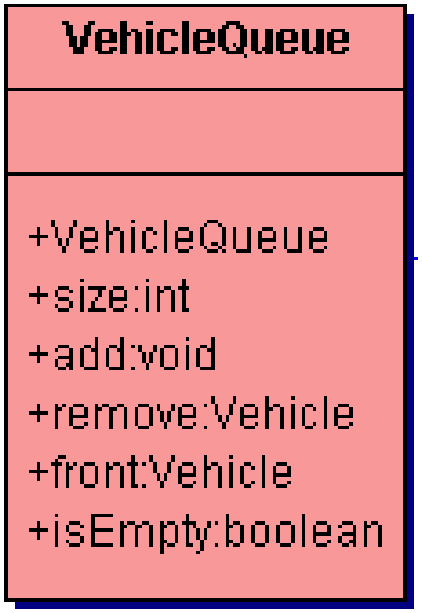
- **View classes** (part of the user interface). These are not part of the backend model.
- **Data consumer or data generator classes**. These are part of the model, though not obvious at first.
- **Helper classes** (code organization/refinement). These are also part of the model, though may not be discovered before testing and implementation begin.

# Helper classes

Two helper classes for this problem:

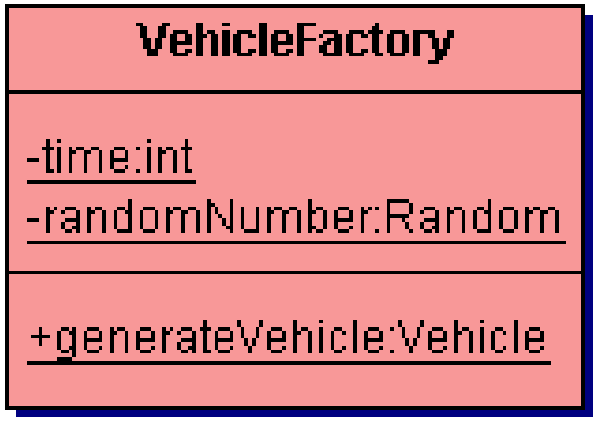
- EventCalendar for the Simulator.
- VehicleQueue to manage vehicles coming onto the highway and to manage tollbooth queues.



# Data generators

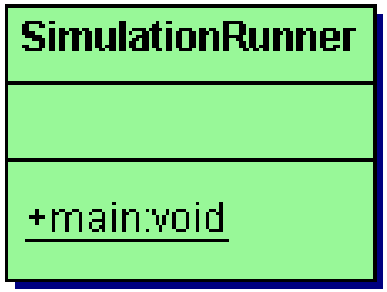
The vehicles have to come from somewhere.  Let's create a VehicleFactory to populate the Highway.

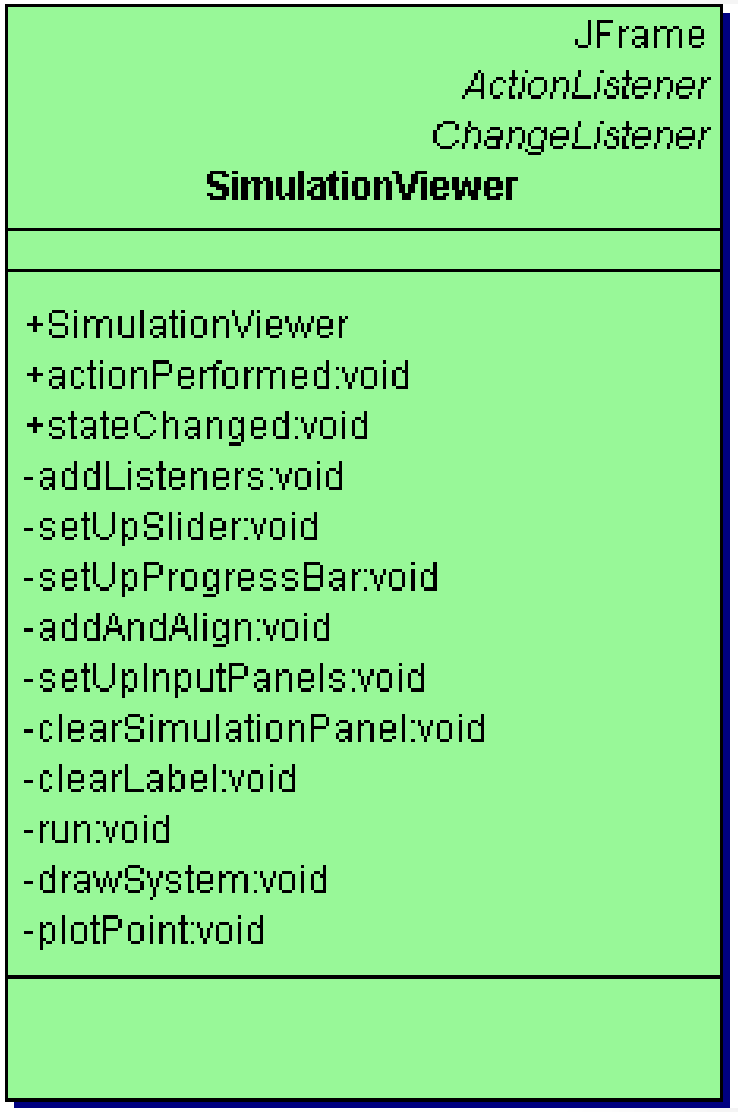


# Viewers

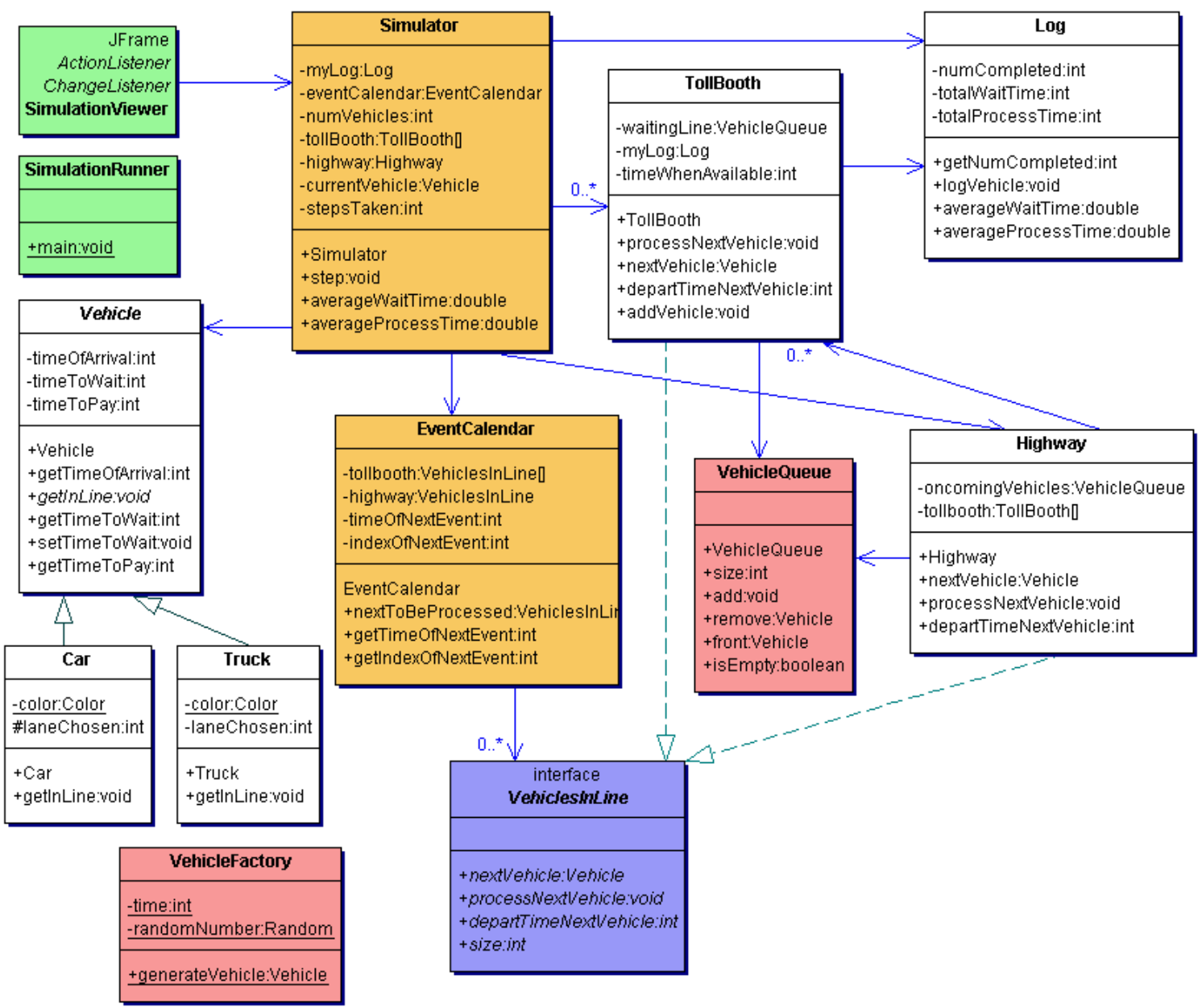
How are you going to view the results of the simulation? How will you feed it information?

- SimulationViewer – GUI
- SimulationRunner – command line viewer



# Putting it all together



# Design Patterns

- **Design pattern** – a general solution to a programming problem that can be applied to develop other solutions.
- Design pattern from the simulator solution: Separate the view from the backend model (Two “views” – SimulationViewer and SimulationRunner.)
- A variation on the **MVC pattern: Model-View-Controller**.

# MVC

- **Model–View–Controller** (MVC) … separates the representation of information from the user's interaction with it. The *model* consists of application data and business rules, and the *controller* mediates input, converting it to commands for the model or view. A *view* can be any output representation of data, such as a chart or a diagram.
- Java implementation:
  - GUI with action listeners – constitute view and controller
  - Classes from the model backend – constitute the model

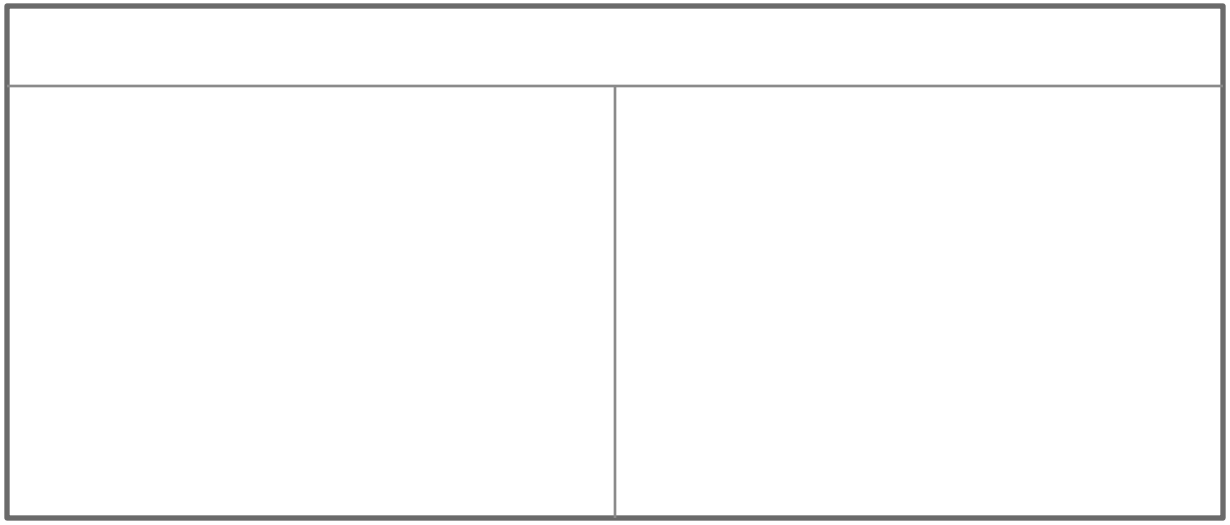
References: Wikipedia MVC, c2.com

# Responsibilities and collaborators

- Each class in the design has a set of **responsibilities** – things it can do.
  - a Car can getInLine.
  - a TollBooth can processNextVehicle in its waiting line
- Each class may require other classes in order to fulfill its responsibilities. These are its **collaborators**.
  - TollBooth collaborators are Highway, Log, and VehicleQueue.
  - VehicleFactory – has no collaborators.
- **Cohesion**: The responsibilities of a class should be closely related to the general purpose of the class. The various class elements should be as closely related as possible.

# Designing via CRC cards

- Design hint #1: Keep it simple.
- How? Spell out only the major responsibilities first.
- Physical representation? 3X5 index card:
  - Class name on the top.
  - Responsibilities on the left.
  - Collaborators on the right.



# Example problem

"ABC Bank wants to upgrade its system for handling checking accounts and customer loans."

# CRC Card - CheckingAccount

"ABC Bank wants to upgrade its system for handling checking accounts and customer loans."

| **CheckingAccount** | |
|---|---|
| **balance**<br>**owner**<br>**deposit**<br>**withdraw**<br>**get balance**<br>**close account**<br>**produce statement** | **Bank customer**<br>**transaction history** |

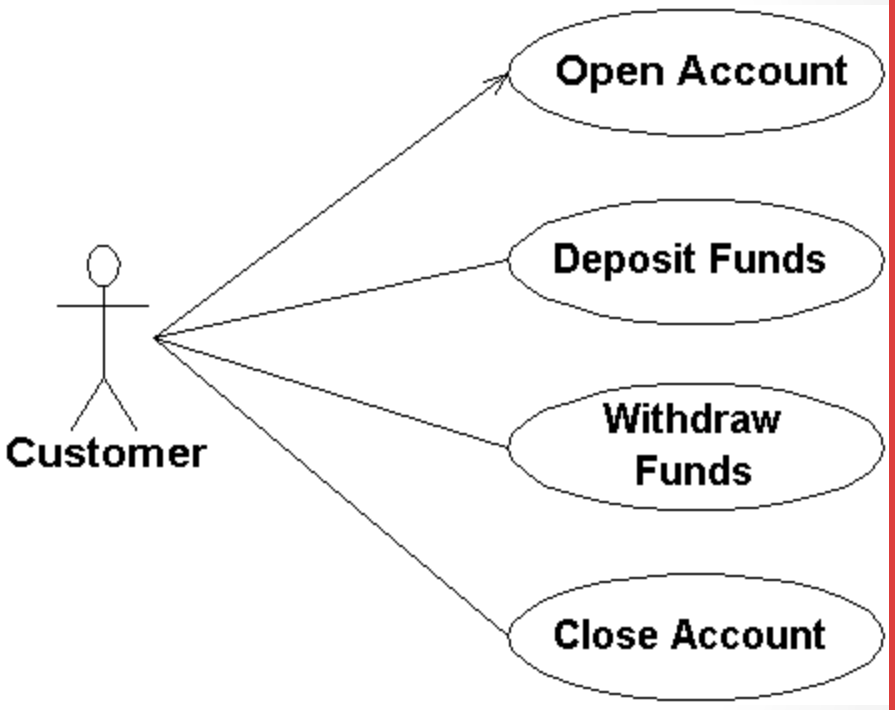
# When are CRC cards useful?

- At the outset of analysis and design
- In small groups of people

CRC cards prevent you from putting too much in a class at the beginning. Classes with too much responsibilities are hard to implement and beasts to maintain. Classes that are too big are sometimes called god classes (anti-pattern).

# Are there other useful methodologies?

- Use cases and use case diagrams.
- Use case = description of how a system provides a service to a stakeholder.
- A use case description consists of a series of steps that define interactions between an "actor" and the system.



# Best Practices: Cohesion

- The responsibilities of a class should be closely related to the general purpose of the class. The various class elements should be as closely related as possible.
- **Cohesion**. The degree to which the members of a class are related to the general purpose of the class.
- Example. Don’t put a customer and checking account into the same class. They are related concepts but they have different purposes.

# Best Practices: encapsulation and information hiding

- **Encapsulation** means gathering together into one package or class all aspects of the entity modeled by that package or class. A consequence of encapsulation is that class data members should be private so they cannot be directly manipulated outside the class.
- **Information hiding** means designing a class so that the implementation details for the class are hidden from other classes.

# Best Practices: loose coupling

- **Coupling**. Dependencies among classes.
- Changes in one class may require changes to another.

- Two extreme kinds of coupling:
  - Internal coupling: one class modifies the data inside another.
  - Parameter coupling: one class needs to use the services provided by another. This is unavoidable in all but the most simple situations.

# Best Practices: Principle of Least Knowledge

- **Principle of Least Knowledge** is a design guideline for developing software… invented at Northeastern University towards the end of 1987, and can be succinctly summarized in one of the following ways:
  - Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
  - …
  - The fundamental notion is that a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents).

Reference: Wikipedia