

Domain Model

- Represent concepts or objects appearing in the problem domain.
 - Also capture object relationships.
- Three types of objects are identified:
 - **Boundary objects**
 - **Entity objects**
 - **Controller objects**

Boundary Objects

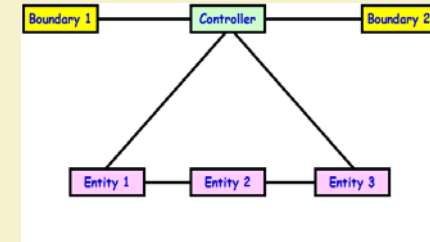
- Handle interaction with actors:
 - **User interface objects**
- Often implemented as screens, menus, forms, dialogs etc.
- Do not perform processing:
 - But may validate input, format output, etc.

Entity Objects

- **Hold information over long term:**
 - e.g. Book, BookRegister
- Normally are dumb servers:
 - **Responsible for storing data, fetching data etc.**
 - **Elementary operations on data: searching, sorting, etc.**
- Often appear as **nouns** in problem description...

- **Overall responsibility to realize use case behavior:**
 - Interface with the boundary objects
 - Coordinate the activities of a set of entity objects
- **Embody most of the business logic required for use case execution:**
- There can be more than one controller to realize a single use case

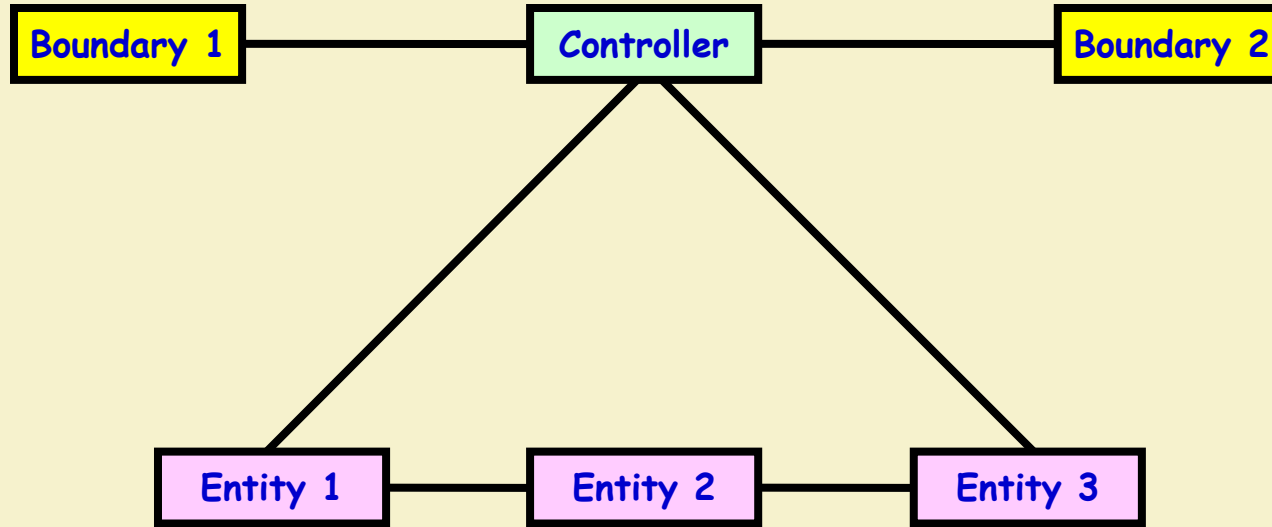
Controller Objects



Controller Classes

- **Controller classes coordinate, sequence, transact, and otherwise control other objects...**
- In Smalltalk MVC mechanism:
 - These were first called controllers

Use Case Realization



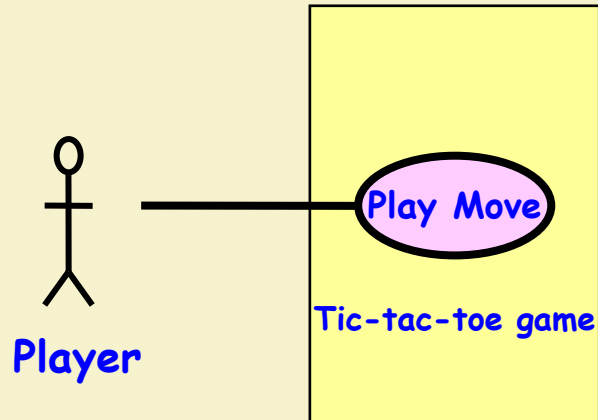
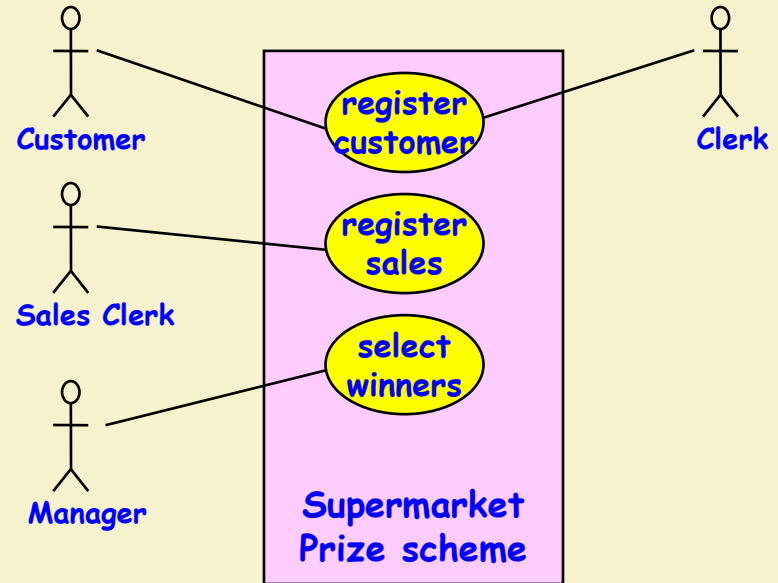
Realization of use case through the collaboration of Boundary, controller and entity objects

Domain Analysis

- Three types of classes are to be identified:
 - Boundary class (Actor-use case pair)
 - Controller class (One per use case)
 - Entity class (Noun analysis)

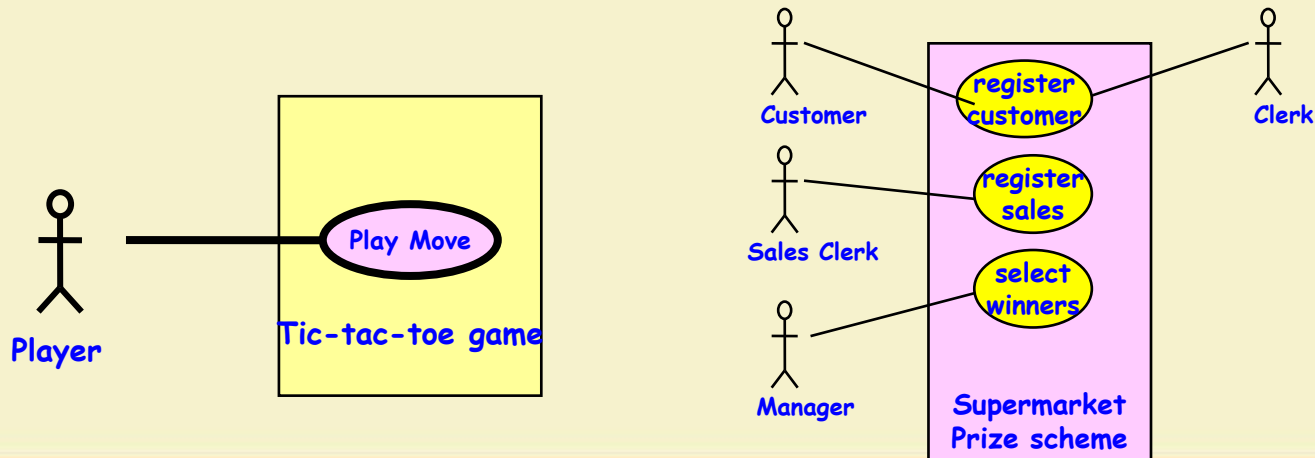
Identification of Boundary Objects from Use Case Diagram

- Need one boundary object :
 - For every actor-use case pair



Identification of Controller Objects

- Examine the use case diagram:
 - Add one controller class for each use case.**
 - Some controllers may need to be split into two or more controller classes if they get assigned too much responsibility.



Identification of Entity Objects by Noun Analysis

- Entity objects usually appear as nouns in the problem description.
- From the list of nouns, need to exclude:
 - **Users (e.g. accountant, librarian, etc)**
 - **Passive verbs (e.g. Acknowledgment)**
 - **Those with which you can not associate any data to store**
 - **Those with which you can not associate any methods**
- Surrogate users may need to exist as classes:
 - **Library member**

Identifying Entity Classes

- Remember that an entity class represents a group (classification) of objects with same behavior.
 - We should therefore look for existence of set of similar objects during noun analysis
- Even then, class names should be singular nouns:
 - Examples: **Book, Student, Member**

A trading house maintains names and addresses of its regular customers. Each customer is assigned a unique customer identification number (CIN). As per current practice, when a customer places order, the accounts department first checks the credit-worthiness of the customer.

Noun Analysis: Example

- A partial requirements document:

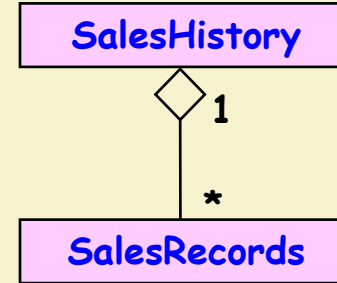
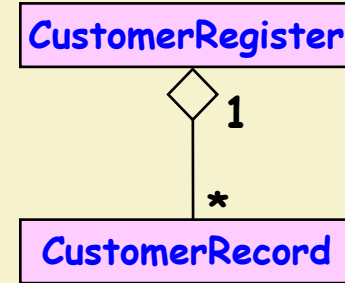
A trading house maintains names and addresses of its regular customers. Each customer is assigned a unique customer identification number (CIN). As per current practice, when a customer places order, The accounts department first checks the credit-worthiness of the customer.

- Not all nouns correspond to a class in the domain model

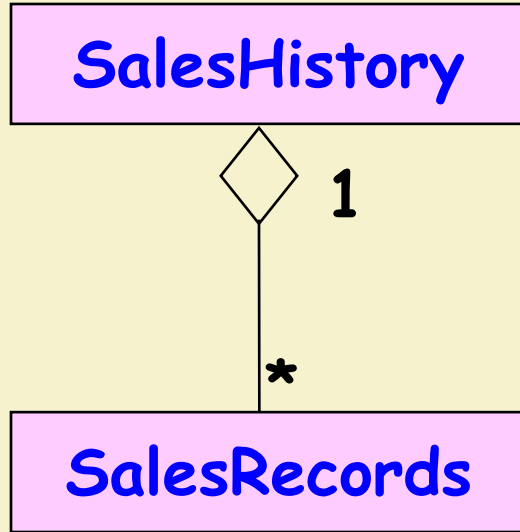
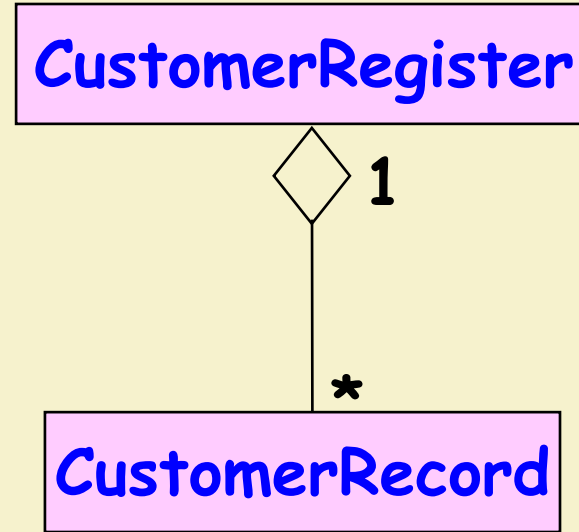
Identifying Classes by Noun Analysis

Identification of Entity Objects: Some Helpful Hints

- Entity Object Usually:
 - Appear as data stores in DFD
 - Occur as group of objects that are aggregated
 - The aggregator corresponds to a register in physical world



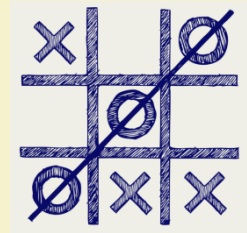
Example Domain Model



- A human player and the computer make alternate moves on a 3X3 square.

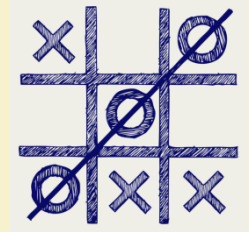
Example 1: Tic-Tac-Toe Computer Game

- A move consists of marking a previously unmarked square.
- The user inputs a number between 1 and 9 to mark a square
- Whoever is first to place three consecutive marks along a straight line (i.e., along a row, column, or diagonal) wins.

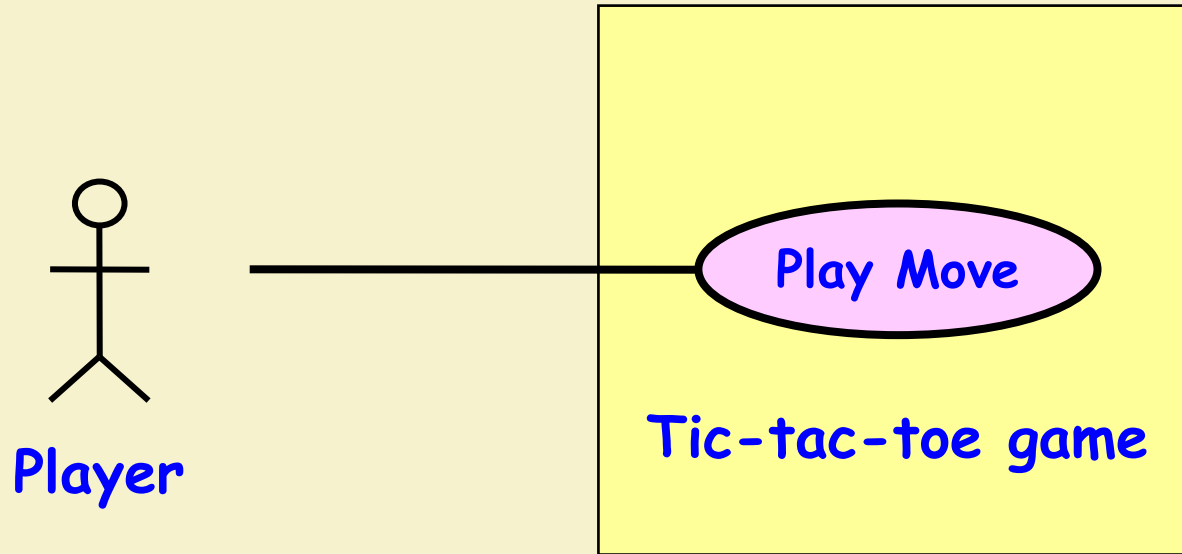


Example 1: Tic-Tac-Toe Computer Game

- As soon as either of the human player or the computer wins,
 - A message announcing the winner should be displayed.
- If neither player manages to get three consecutive marks along a straight line,
 - And all the squares on the board are filled up,
 - Then the game is drawn.
- The computer always tries to win a game.



Example 1: Tic-Tac-Tie Use Case Model



Example 1: Initial and Refined Domain Model

Board

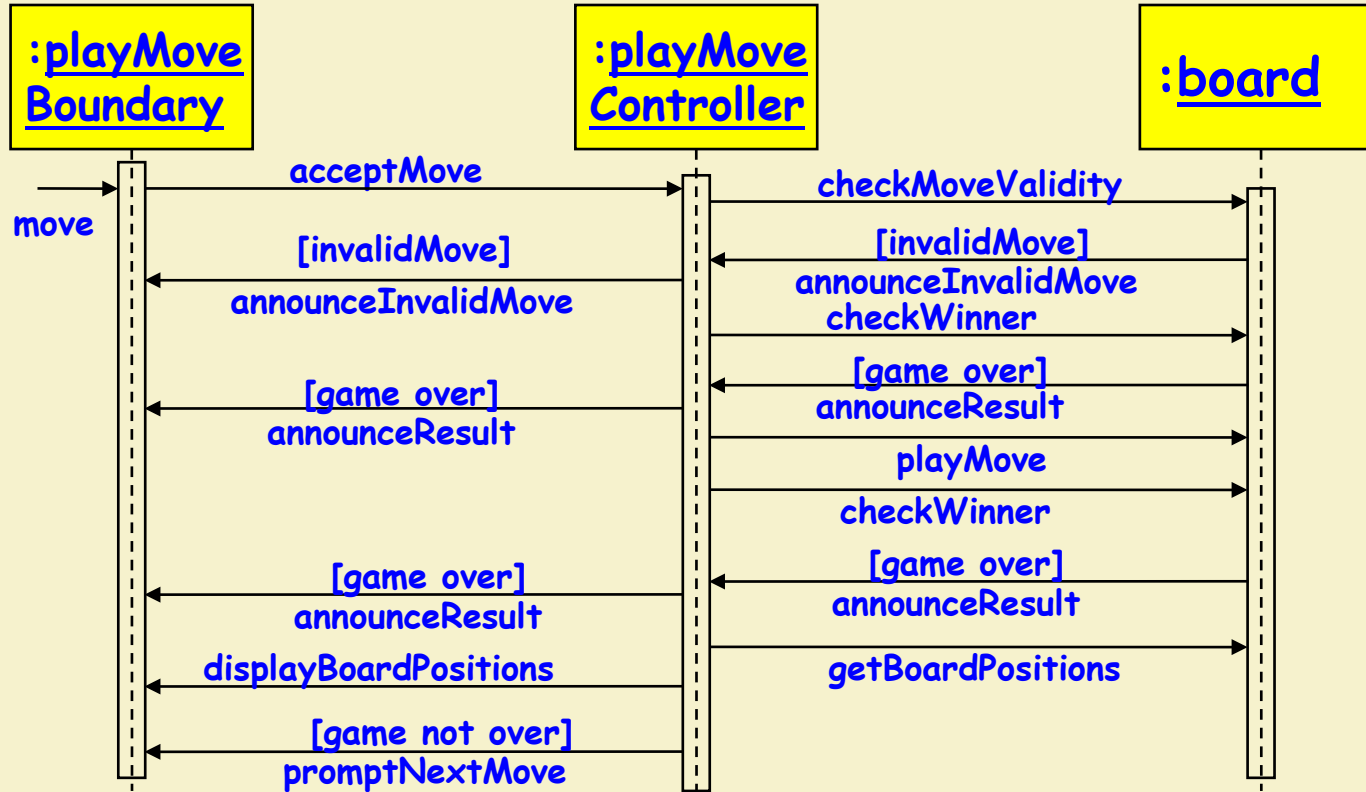
Initial domain model

PlayMoveBoundary

PlayMoveController

Board

Refined domain model

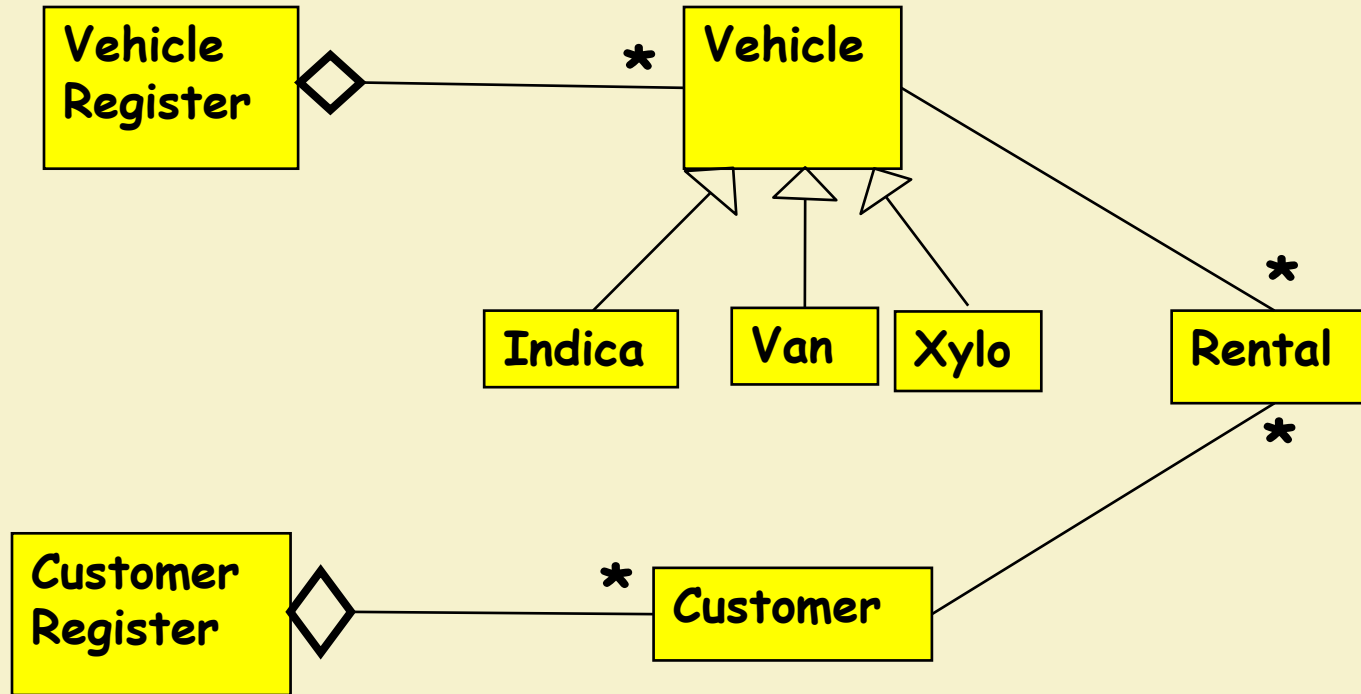


**Example 1:
Sequence
Diagram for
the play
move use
case**

- The fleet of vehicles at a travel agency consists of vehicles of the types Tata Indica, Maruti van, and Mahindra Xylo.
- Customers first register by providing address details and depositing Rs.5000/- with the agency.
- The details of the customers such as the name, address, and phone number are maintained by the agency in a customer register.
- The regular customers of the travel agency can rent any vehicle they want. The details of the travel, date, distance, AC/NonAC are entered by the driver.

**Exercise 1:
Construct
Domain
Model**

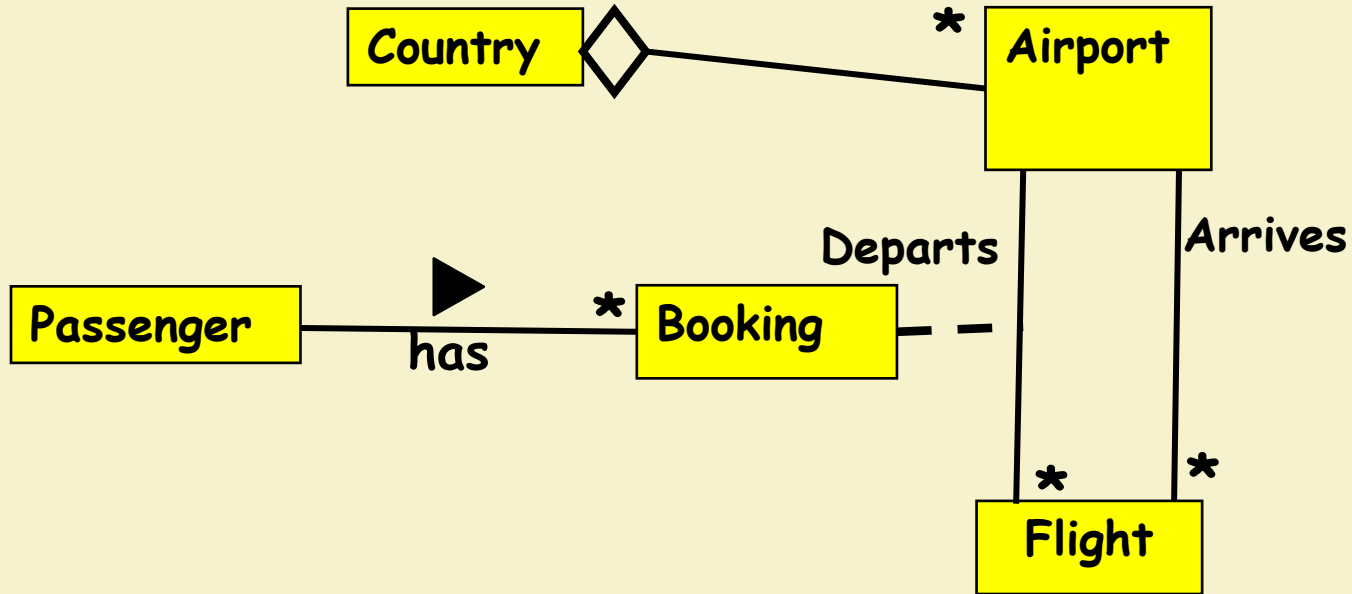
Solution



Exercise 2

- A country has many airports.
- A set of flights arrive and depart from each airport.
- Each flight is characterized by its flight number, origin and destination airports, scheduled times at the origin and destination, and days of week in which it operates.
- A passenger can book a flight for a specific date. Each passenger has a name, a mobile number, and address.
- A flight booking is characterized by a transaction number, ticket Price, and any special amenities booked (such as preordered meals).

Solution



- CRC Card is used to assign responsibilities (methods) to classes.
- Complex use cases:
 - These are realized through collaborative actions of dozens of classes.
 - **Without CRC cards, it becomes difficult to determine which class should have what responsibility...**

Class-Responsibility-Collaborator(CRC) Cards

- Pioneered by Ward Cunningham and Kent Beck.
- Index cards prepared one each per class.
- Contains columns for:

—Class responsibility

—Collaborating objects

Class name	
Responsibility	Collaborator

CRC Cards Cont...

- **Systematize development of interaction diagram for complex use cases.**
- Team members participate to determine:
 - The responsibility of classes involved during a use case execution

CRC Cards

- **Responsibility:**
 - Method to be supported by the class.
- **Collaborator:**
 - Class whose service (method) would have to be invoked

Class name	
Responsibility	Collaborator

An Example: CRC Card for the BookRegister class

BookRegister	
FindBook	Book
CreateBook	Book
Reserve	Book

Class name

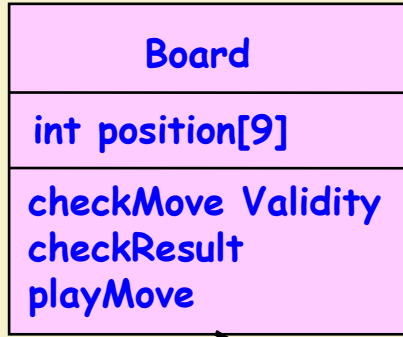
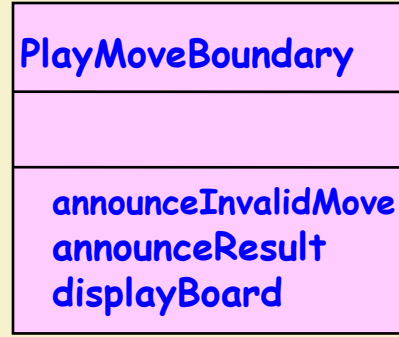
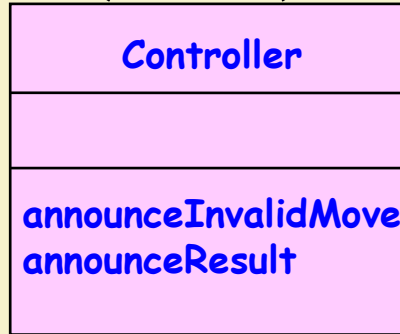
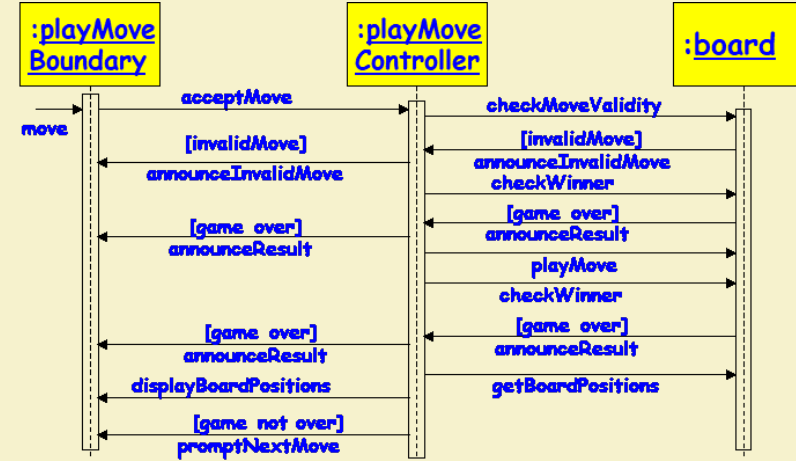
Responsibility

Collaborator



Using CRC Cards

- After developing a set of CRC cards:
 - Run structured walkthrough scenarios
- **Walkthrough of a scenario :**
 - A class is responsible to perform some responsibilities
 - It may take help of a collaborator --- another class
 - You may discover missing responsibilities and classes



Example 1: Class Diagram

- Supermarket needs to develop software to encourage regular customers.
- Customer needs to supply his/her:
 - Residence address, telephone number, and the driving licence number.
- Each customer who registers is:
 - Assigned a unique customer number (CN) by the computer.

**Example :
Supermarket
Prize Scheme**

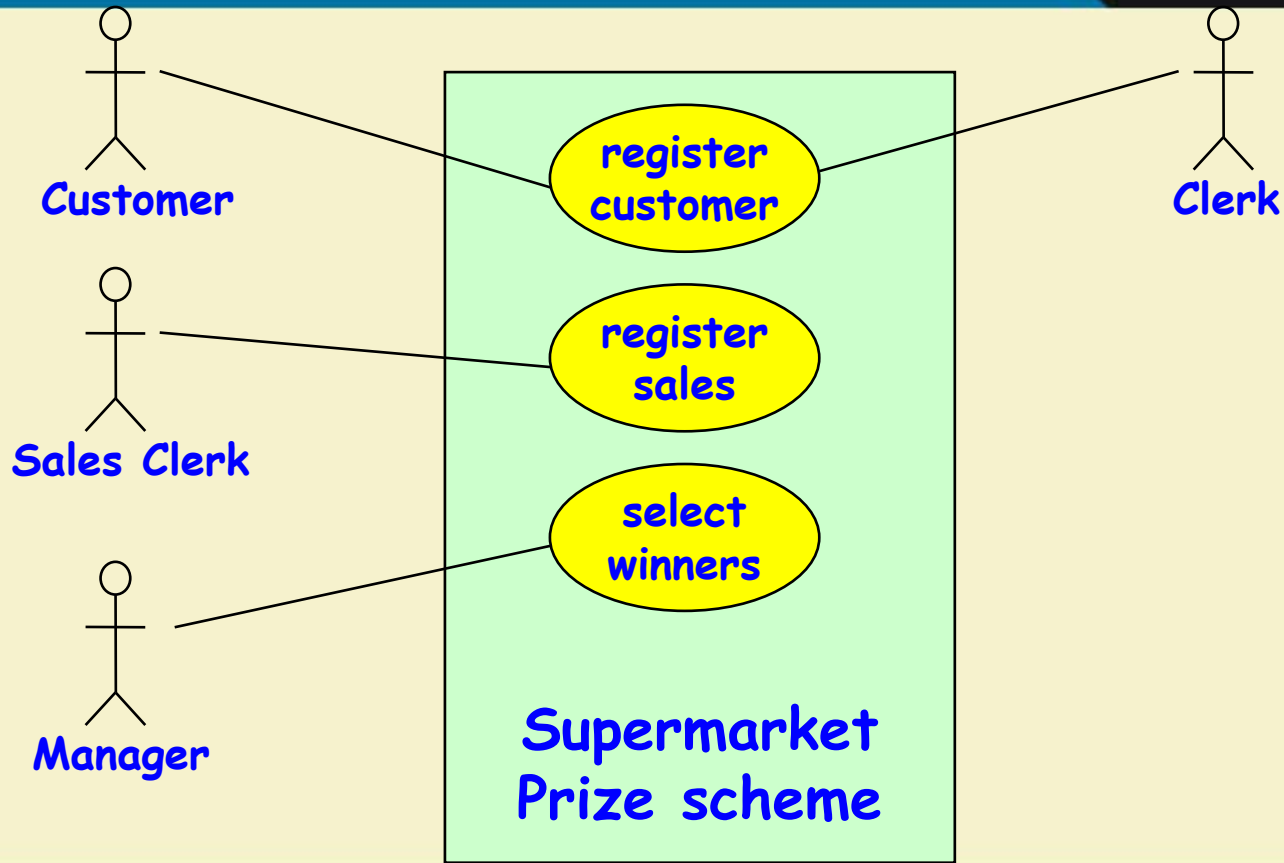


**Example:
Supermarket
Prize Scheme**

- A customer can present his CN to the staff when he makes any purchase.
- The value of his/her purchase is credited against his/her CN.
- At the end of each year:
 - The supermarket awards surprise gifts to ten customers who make highest purchase.

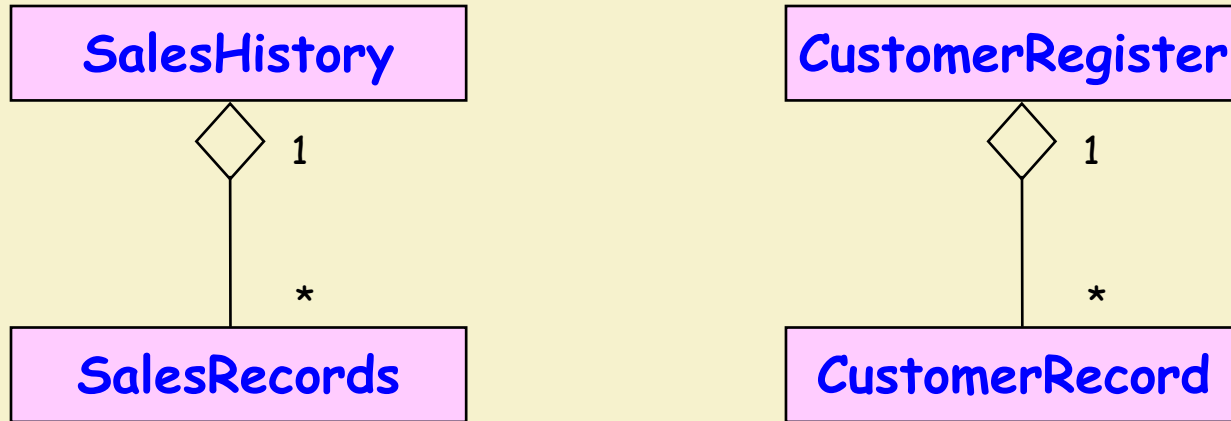
Example: Supermarket Prize Scheme

- It also, awards a 22 carat gold coin to every customer:
 - Whose purchases exceed Rs. 10,000.
- The entries against the CN are reset:
 - On the last day of every year after the prize winner's lists are generated.

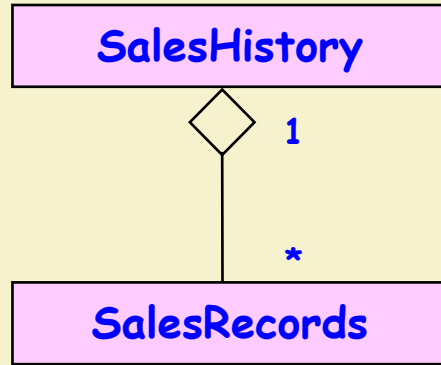
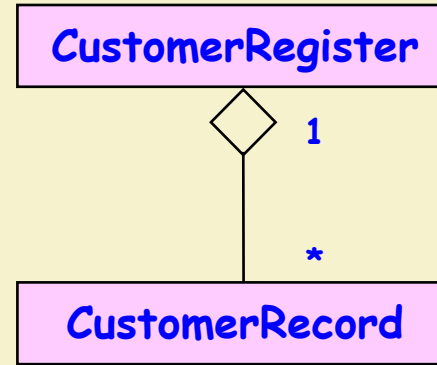


Solution: Use Case Model

Solution: Initial Domain Model



Initial domain model



**Solution:
Refined
Domain
Model**

RegisterCustomerBoundary

RegisterCustomerController

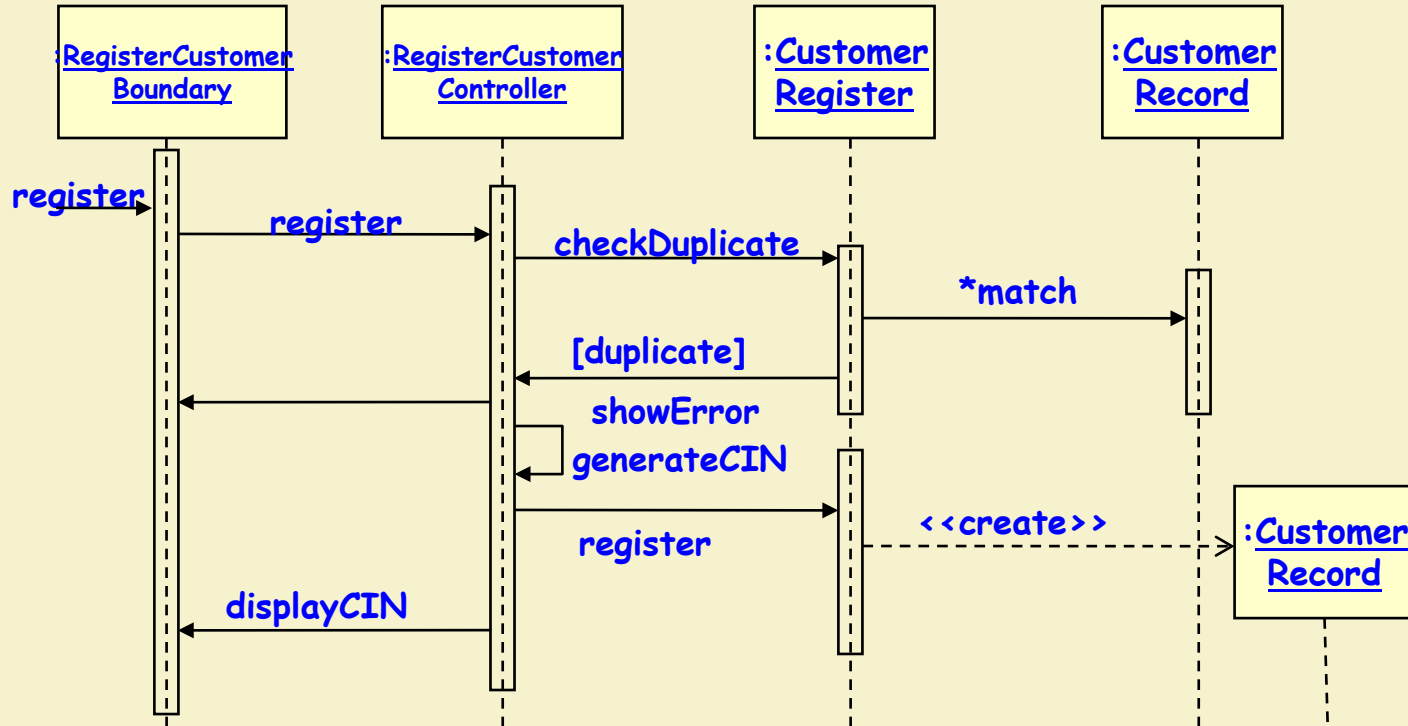
RegisterSalesBoundary

RegisterSalesController

SelectWinnersBoundary

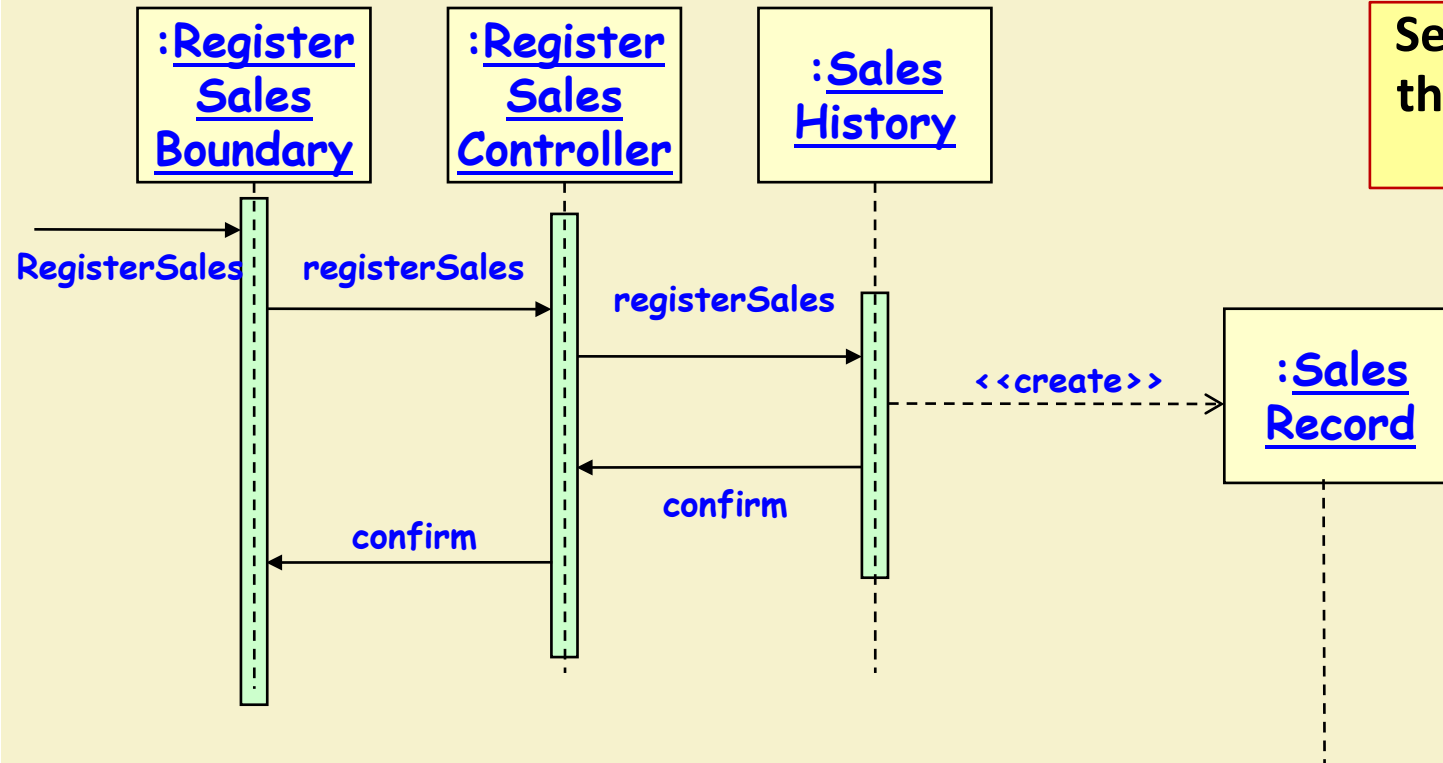
SelectWinnersControllers

Sequence Diagram for the Register Customer Use Case



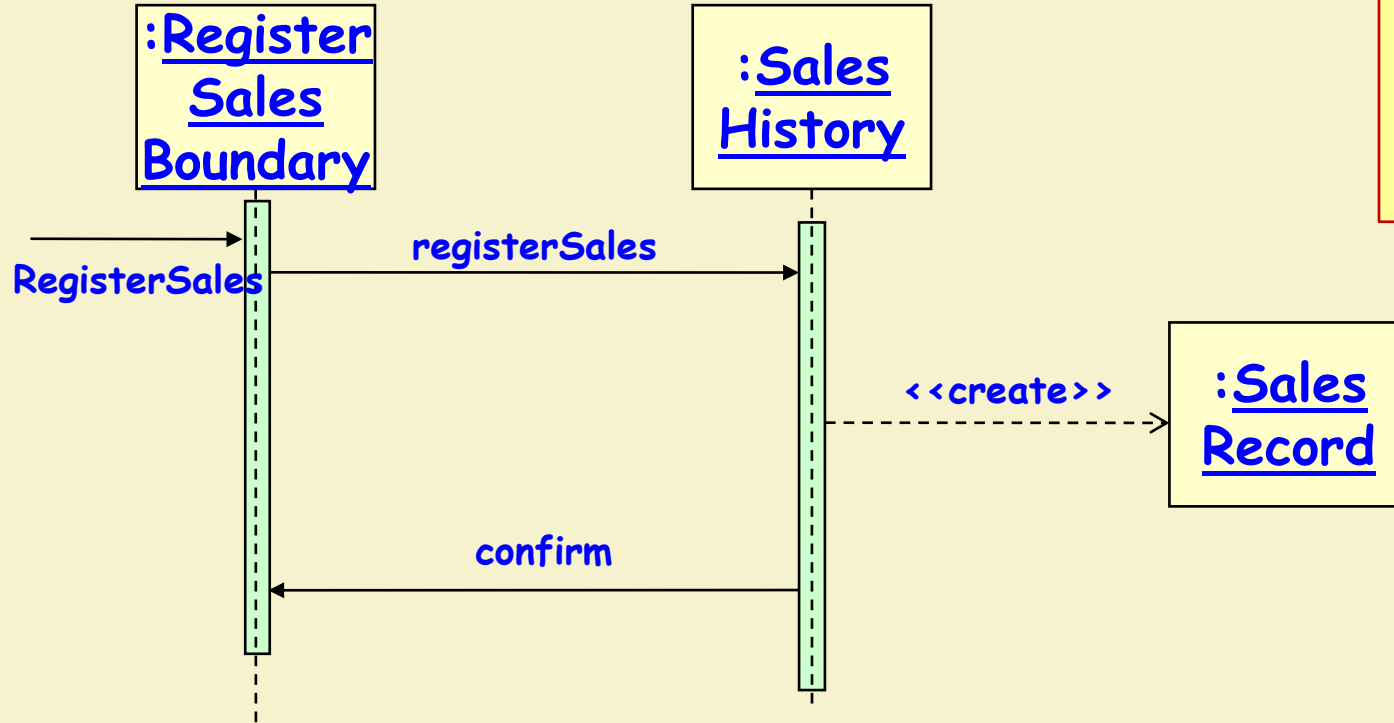
Sequence Diagram for the register customer use case

Sequence Diagram for the Register Sales Use Case



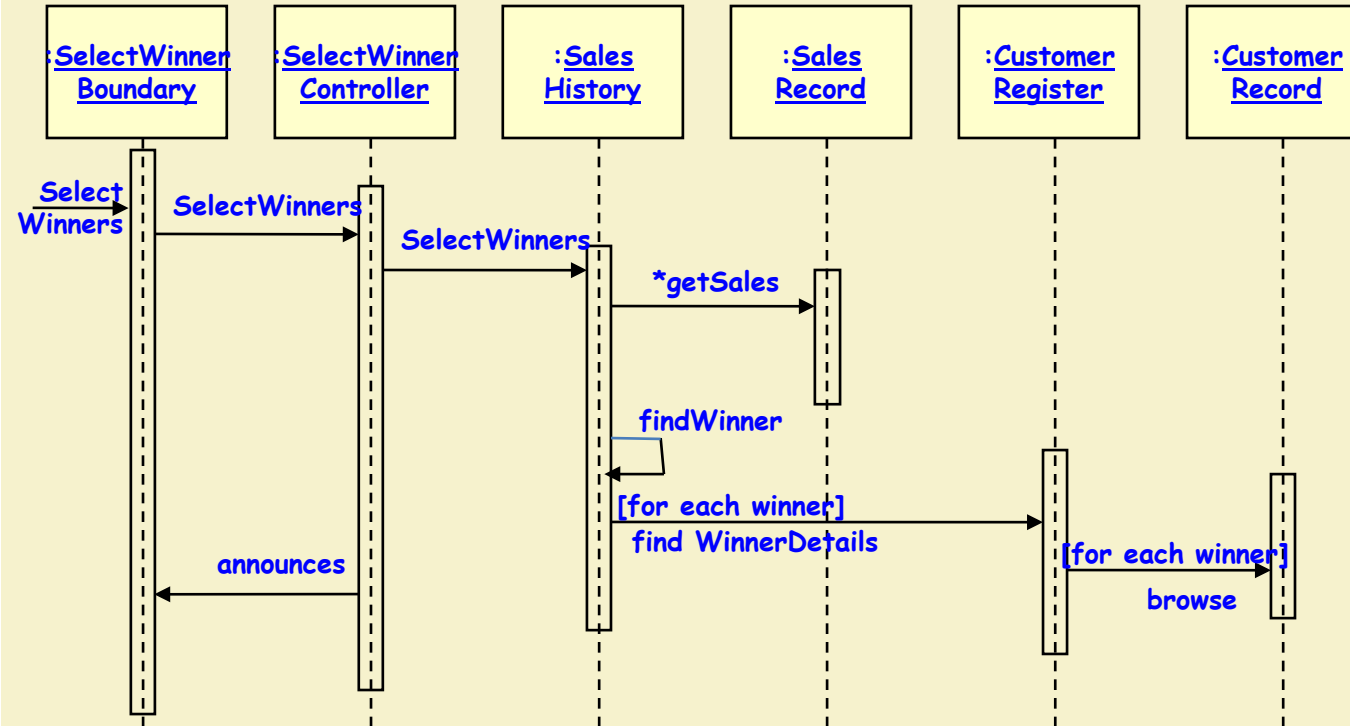
Sequence Diagram for the register sales use case

Refined Sequence Diagram for the Register Sales Use Case



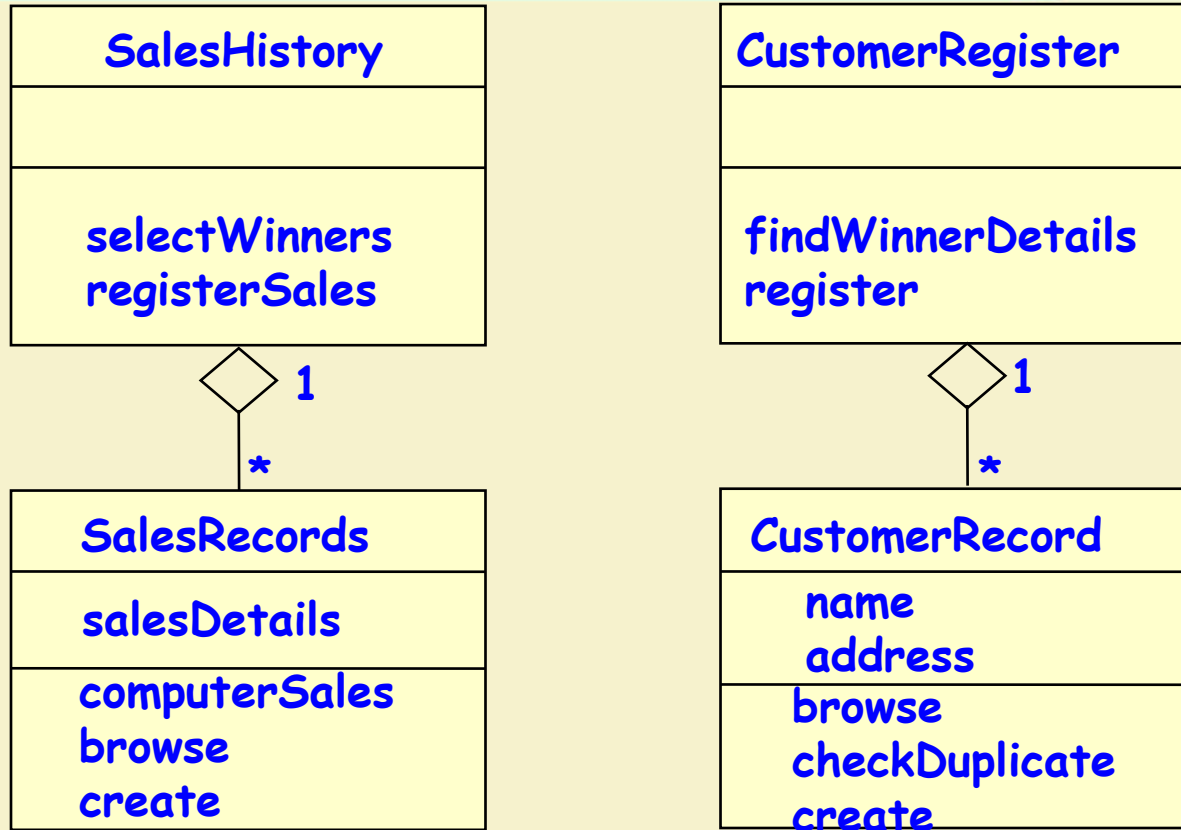
Refined Sequence Diagram for the register sales use case

Sequence Diagram for the Select Winners Use Case



Sequence Diagram for the select winners use case

Class Diagram



Video Rental Store Software

- A video rental store has a large collection of video CDs and DVDs in VHS and MP4 format as well as music CDs as loanable items.
- A person can become member a member by depositing Rs. 1000 and filling up details such as name, address, and telephone number. A member can cancel his membership and take back his deposit, if he has no dues outstanding against him.
- Whenever the store purchases a new item, details such as price and date of procurement are entered. The daily rental charge is also entered by the manager. After passage of a year, the daily rental charge is automatically halved.
- A member can, at a time, take on loan at most three loanable items. The details are entered by a store clerk and a receipt indicating the daily rental charge should be printed by the software.
- Whenever a member returns his loaned item(s), the due amount to be paid is displayed. After the amount is paid, the items are marked returned.
- If a customer loses or damages any item, the full price of the item is charged to him and the item is removed from the inventory.
- If an item is not lent out by anyone for even once over a year, the item is sold at 10% of the purchase price and is removed from the inventory.
- The manager can, at any time, check the profit/loss account.

Personal Library System

- The software is required to manage the collection of books by individuals.
- A person can have a few hundreds of books. The details of all the books such as name of the book, year of publication, date of purchase, price, and publisher must be entered. A book is to be given a unique serial number by the computer which would be written by the owner using a pen on the book. Before a friend can be lent a book, he must be registered. The registration data would include name of the friend, address, land line number, and mobile number. Before a friend is issued a book, the various books outstanding against him also with the date borrowed are displayed. The date of issue and the title of the book are stored. When a friend returns a book, the date of return is stored and the book is removed from his borrowing list. Up on query, the software should display the name, address, and telephone numbers of each friend against whom books are outstanding along with the titles of the outstanding books and the date on which those were issued.
- The owner of the library software, when he borrows books from his friends, would enter the details regarding the title of books borrowed, and the date borrowed. Similarly, the return details of books would be entered. The software should be able display all the books borrowed from various friends. The owner should be able to query about the availability of a particular book, the total number of books in the personal library,

- The staff and students of IIT register their vehicle by filling up a form which a security staff would enter into the system.
- Each time a vehicle enters or leaves the campus, a camera reads the registration number of the car, and the model of the car. If the car is registered, the check gate should lift automatically to let in (or out) the vehicle. The details regarding the entry and exit of the vehicle are registered in the database. For outside vehicles, the driver needs to enter the purpose of entry, the model number, the registration number, and a photograph of the vehicle is stored in the database.
- When an outside vehicle leaves the campus, the details are automatically registered in the database. For any external vehicle that is inside the campus for more than 8 hours, the driver is stopped by the security personnel manning the gate, queried to satisfaction, and the response are recorded. Considering that there have been several incidence of speeding and rough driving, the security personnel are empowered to telephone the registration number of errant vehicles to the main gate. For outside vehicles, the driver is quizzed at the check gate and his future entry is barred if the response is not satisfactory. For inside vehicles, a letter is issued to the dean to deal with the employee or student as the case might be.
- The security officer can check the data regarding the number of vehicles going in and coming out of the campus (over a day, month or year), the total number of vehicles currently inside the campus.

Students' Auditorium Management Software

- Various types of social and cultural events are conducted in the student auditorium. There are two categories of seats: balcony and ordinary seats. The maximum number of balcony seats that can be booked per ticket is 2.
- The show manager fixes the prices of the two categories of seats for a specific show, depending on the popularity of a show.
- The show manager also enters the show dates, the number of shows on any particular date and the show timings. It is expected that the software would support a functionality to let the show manager configure the different show parameters.
- The system should let the spectators query the availability of different classes of seats for a show on-line. A spectator can book a seat for a single show only through regular cash payment and the software should print out the ticket showing the seat numbers allocated. A spectator should be able to cancel his booking before 3 clear days of the show.

Object-Oriented Design Principles



Symptoms of Rotten Code

- **The system is rigid:**
 - Hard to change anything because for making one change, need to change everything.
- **The system is fragile:**
 - Changes cause the system to break at unexpected places.
- **The system is immobile:**
 - Not reusable.
- **The system is viscous:**
 - **The system is opaque** —hard to understand.
 - The system is needlessly complex.
 - The system contains needless repetition.

Main Culprit:
Dependencies in
Design

- Dependencies can arise among classes, packages, and subsystems:
 - Due to a large number of reasons
- Unless dependencies are kept to a minimum:
 - Design and code become unmaintainable
 - Even small modifications break the system
 - Scope of reuse diminishes
 - Too many bugs and test/debug costs rise

Common Causes of Dependency

- Tight coupling
- Overuse of inheritance
 - **Ex: Extending functionality by subclassing**
- Creating an object by explicitly specifying a class.
- Dependence on specific operations
- Dependence on hardware and software platforms

SOLID Principles to Avoid Poor Design

- Acronym of 5 acronyms:
 - **SRP:** Single Responsibility Principle
 - **OCP:** Open-Closed Principle
 - **LSP:** Liskov Substitution Principle
 - **ISP:** Interface Segregation Principle
 - **DIP:** Dependency Inversion Principle

The Open/Closed Principle (OCP)

based on an article of that title
by Robert Martin

- **Classes should be open for extension, but closed for modification.**
 - **Open:**

A class is open if it can be extended:

 - Add data members and operations through inheritance.
 - **Closed:**

A class is closed if it is available for use by other components but is not itself changed:

 - **e.g., it can be put under a configuration control tool, allowing read only access.**
 - **Declaring it final etc.**

Main Concept

- The open/closed principle states:
 - **Once components are successfully tested**
--- they should never change, only
extended to meet the requirements
changes.

Solution

Program to an Interface.

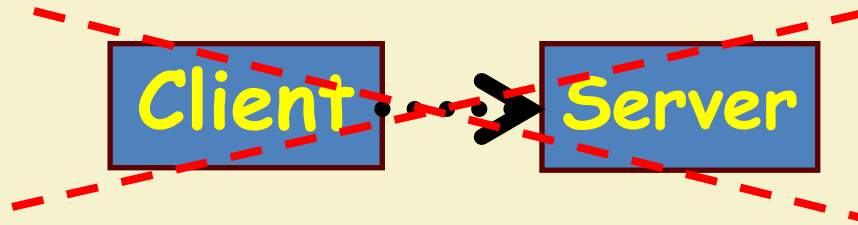
- If you are not changing working code:
 - You are unlikely to break it.
- Define an abstract or interface class:
 - Extend the interface/abstract class whenever any new feature is required.
 - The client class selects the appropriate object through dynamic binding.

Key Design Idea of OCP

- **Open:**
 - Can add functionality or data by extending it.
- **Close:**
 - Can not modify it.
 - Its protocol available for usage by other classes does not change.
 - That is, it has a stable interface
 - Normally implemented with abstract classes

The Role of Interface/Abstraction

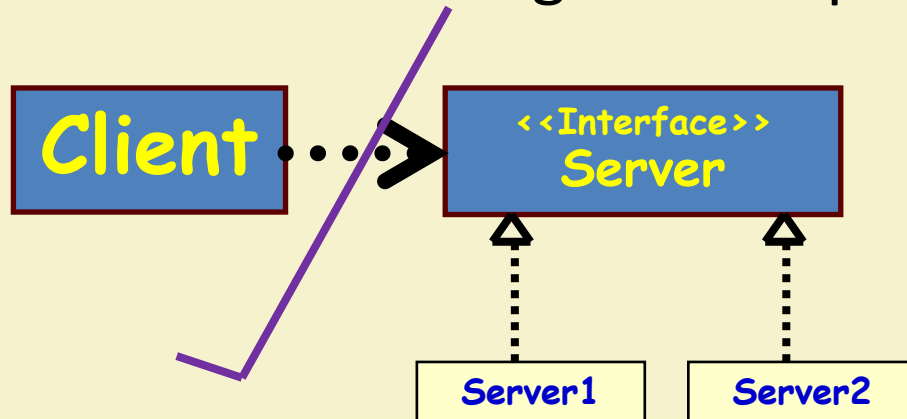
- It is very difficult to build a class that would never be required to change.
- Two major situations under which changes to a class becomes unavoidable:
 - **Latent errors force change.** We can't fix incorrect operation by extension. The component itself must be fixed.
 - **Performance problems force change.** Met usually by changing a computational algorithm or data structure.
- **Changes due to other reasons:**
 - Can be achieved through extension.



OCP: Client-Server Classes

- OCP violation:**

- Server can change and requires changes to the client.



OCP
Compliant

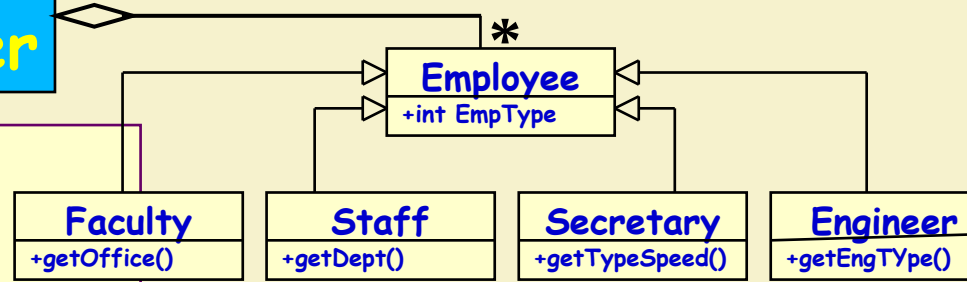
Open/Closed Principle (OCP): Main Concept

- The open/closed principle (OCP) states:
 - **Once components are successfully tested --- they should never change, only extended to meet the requirements changes.**

- **Use of interfaces/Abstract Classes is a key idea in the Open/Closed Principle.**
 - Interfaces are implemented, but are themselves closed to modification.
 - **Since Interfaces have no implementation, they have no latent errors to fix and no performance issues.**

Example: Naïve Solution

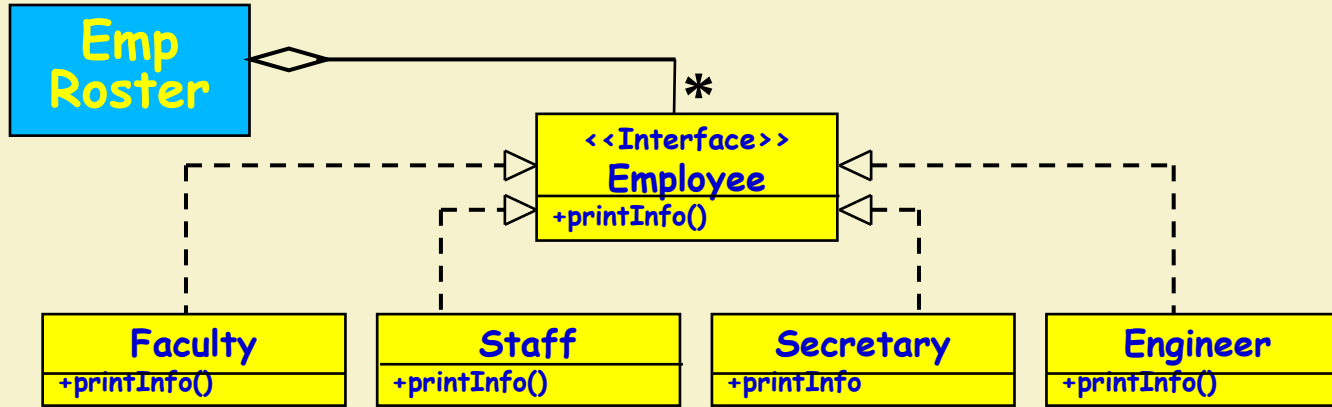
Emp Roster



```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        if (emps[i].empType == FACULTY)
            printfFaculty((Faculty)emps[i]);
        else if (emps[i].empType == STAFF)
            printfStaff((Staff)emps[i]);
        else if (emps[i].empType == SECRETARY)
            printfSecretary((Secretary)emps[i]);
    }
}
```

What if we
need to add
Engineer??

Example 1: OCP Solution



```
void printEmpRoster(Employee[] emps) {
    for (int i; i<emps.size(); i++) {
        emps[i].printInfo();
    }
}
```

When Engineer is added, printEmpRoster() does not even need to recompile.
PrintEmpRoster() is open to extension, closed for modification.

OCP: Summary

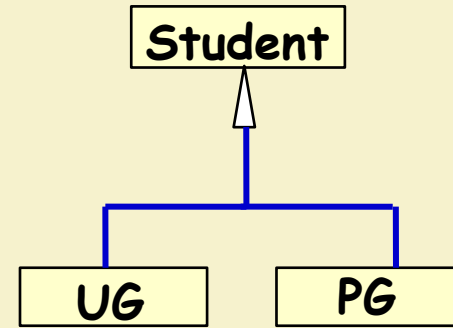
- The Open/Closed Principle states:
 - Well designed code can be extended and not modified.
 - **New features are added by adding new code rather than changing already working code.**
- Software to be reusable, maintainable, and robust:
 - Must be extensible without requiring change.
 - **Use of interface classes is the key**

Liskov Substitution Principle (LSP)

Ref: Barbara Liskov, “Data Abstraction and Hierarchy”, ACM SIGPLAN Notices, 23, 5 (May 1988)

- **Statement: “An instance of a derived class should be able to replace any instance of its super class”**
- In effect, the base class establishes a basic behavior (functionality) that clients know of.
 - **If a derived class violates the contract, then a violation of LSP occurs.**

Liskov Substitution Principle

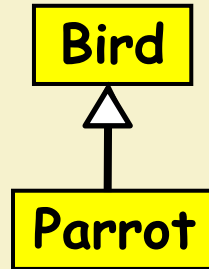


In Simple Words

- Any subclass object should always be seamlessly usable in place of its parent class object.
- **Central idea** - All derived classes must honour the contracts of their base classes:
 - IS A ➔ same public behavior

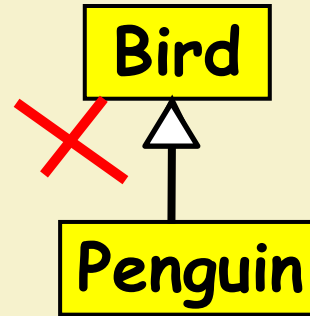
Inheritance Appears Simple

```
class Bird {           // has beak, wings,...  
    public void fly(); // Bird can fly  
};  
class Parrot extends Bird { // Parrot is a bird  
    public void mimic(); // Can Repeat words...  
};  
  
// ...  
Parrot mypet;  
mypet.mimic(); // my pet being a parrot can Mimic()  
mypet.fly();   // my pet "is-a" bird, can fly
```



```
class Penguin extends Bird {  
    public void fly() {  
        error ("Penguins don't fly!"); }  
};  
  
void PlayWithBird (Bird abird) {  
    abird.fly();    // OK if Parrot.  
    // if bird happens to be Penguin...OOOPS!!  
}
```

Penguins Do Not Fly!



- Does not model: "Penguins can't fly"
- It models "Penguins may fly, but if they try it is error"
- Run-time error if attempt to fly → not desirable
- Check Substitutability - **Fails LSP**

Summary

- The Liskov Substitution Principle (LSP) states:
 - **Every function that operates on a base class reference should operate successfully when a derived class object is substituted for the base object.**
 - In doing this it should not need to know which specific derived object.

The Single Responsibility Principle (SRP)

based on an article of that title
by Robert Martin

Single Responsibility Principle

- Each responsibility is a reason to change a class:
 - **A Class should have only one reason to change.**
- How to know whether a class has many responsibilities?
 - The class's responsibility is described using several “and”s
- SRP violation Reduces class cohesion

**Single-
Responsibility
Principle (SRP)**

- Also referred to as “modularity”
- If a class has two responsibilities:
 - The code for these get coupled, and often need to change together.
- How to solve this problem?
- **Solution hint: delegate!**

```
public interface Worker {  
    public void eat();  
    public void work();  
}
```

```
public class OfficeWorker implements  
    Worker{  
    public void work() { // ....work 8 hrs}  
    public void eat() { // .... eat in lunch  
        break}  
}
```

SRP: Example

```
public class RobotWorker implements  
    Worker {  
    public void work() {  
        // Do work }  
  
    public void eat() {  
        throw new  
            NotImplementedException();  
    }  
}
```

How to add a robot?

SRP – Fix!

```
public interface Worker {
```

```
    public void work();
```

```
}
```

```
public interface Eater{
```

```
    public void eat();
```

```
}
```

```
public class OfficeWorker implements Worker, Eater{  
    ... }
```

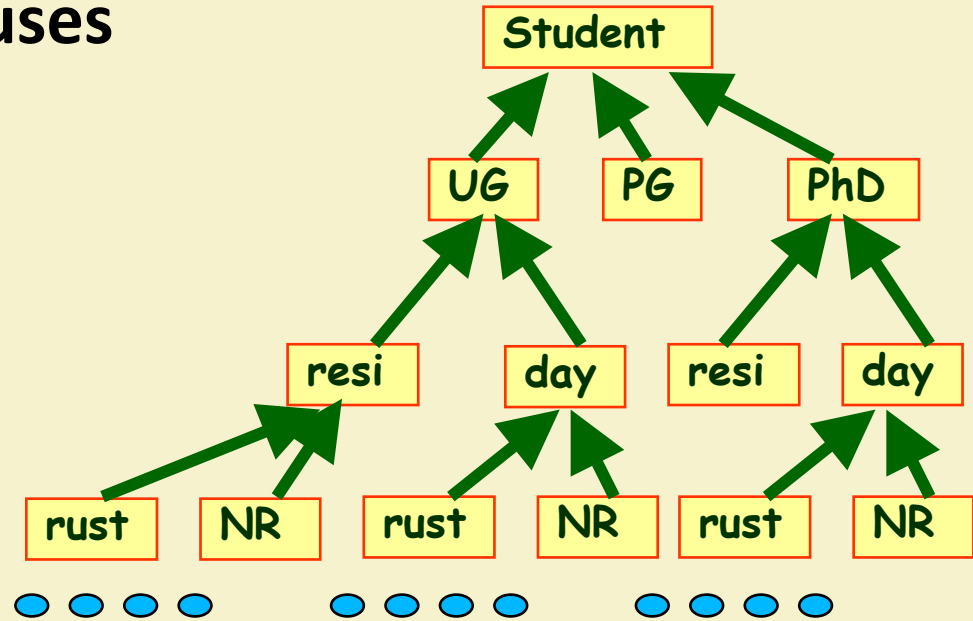
```
public class RobotWorker implements Worker{ ...}
```

Now each interface has
only one purpose

SRP: A Reflection on Causes

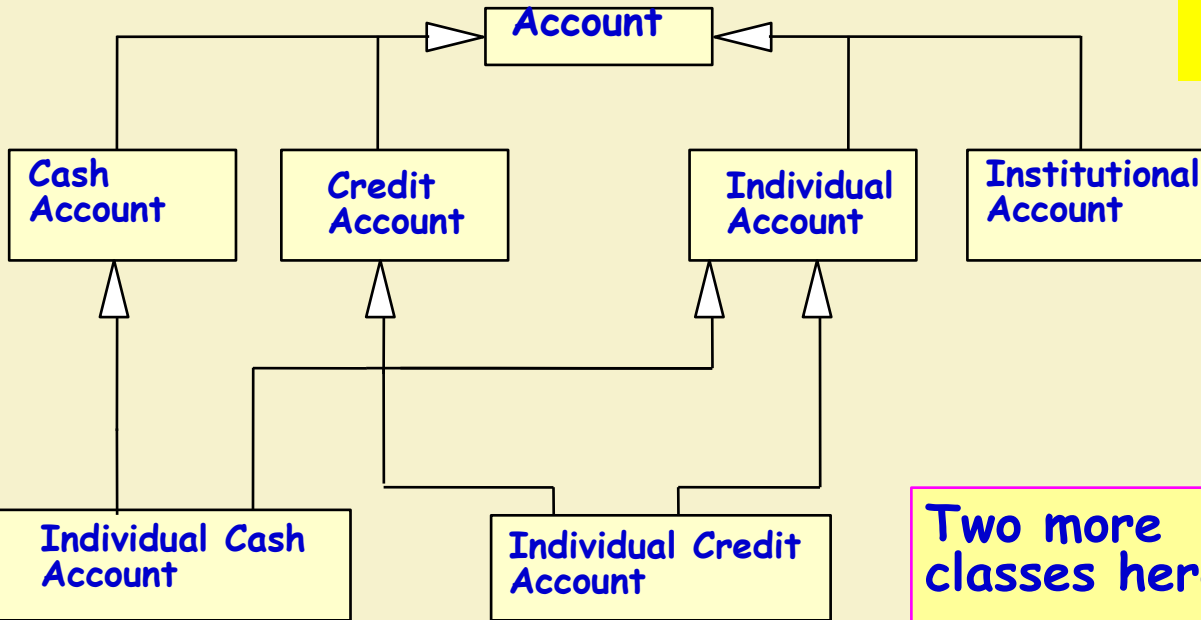
- Often inheritance is used to extend a class along unrelated dimensions:

- Reduces method cohesion and violates SRP



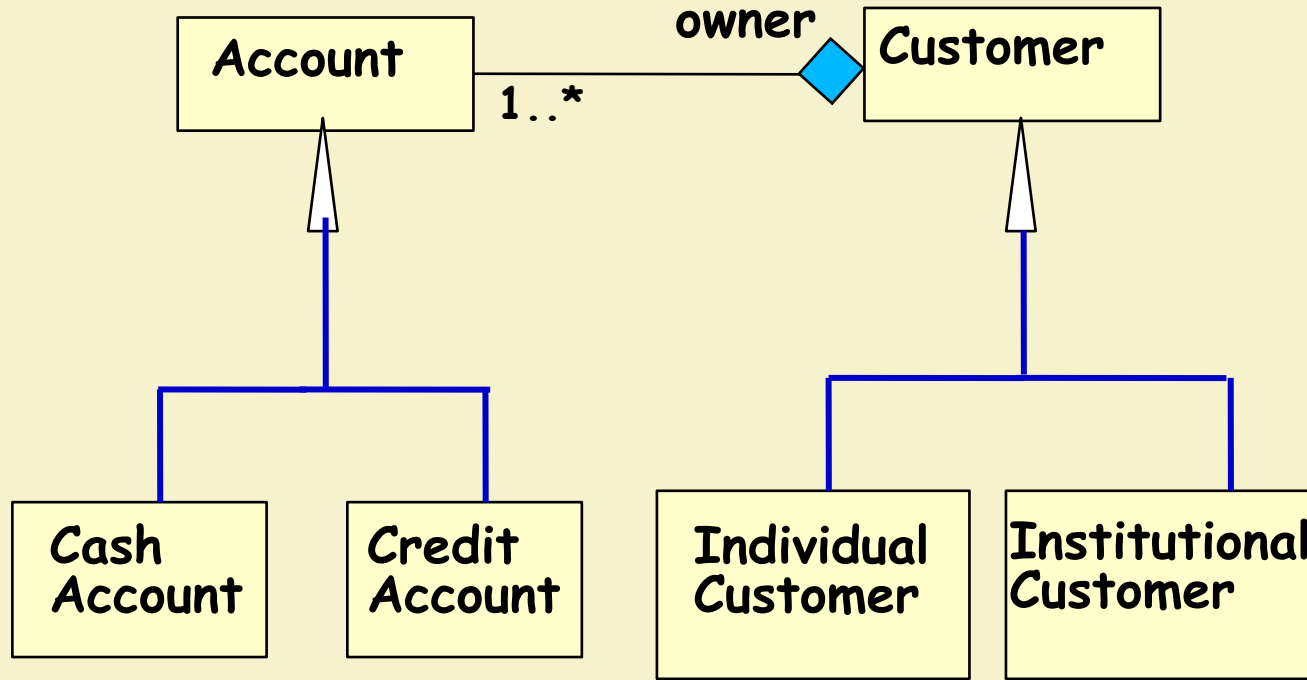
Naïve Use of Inheritance

Do not inherit
unrelated dimensions!

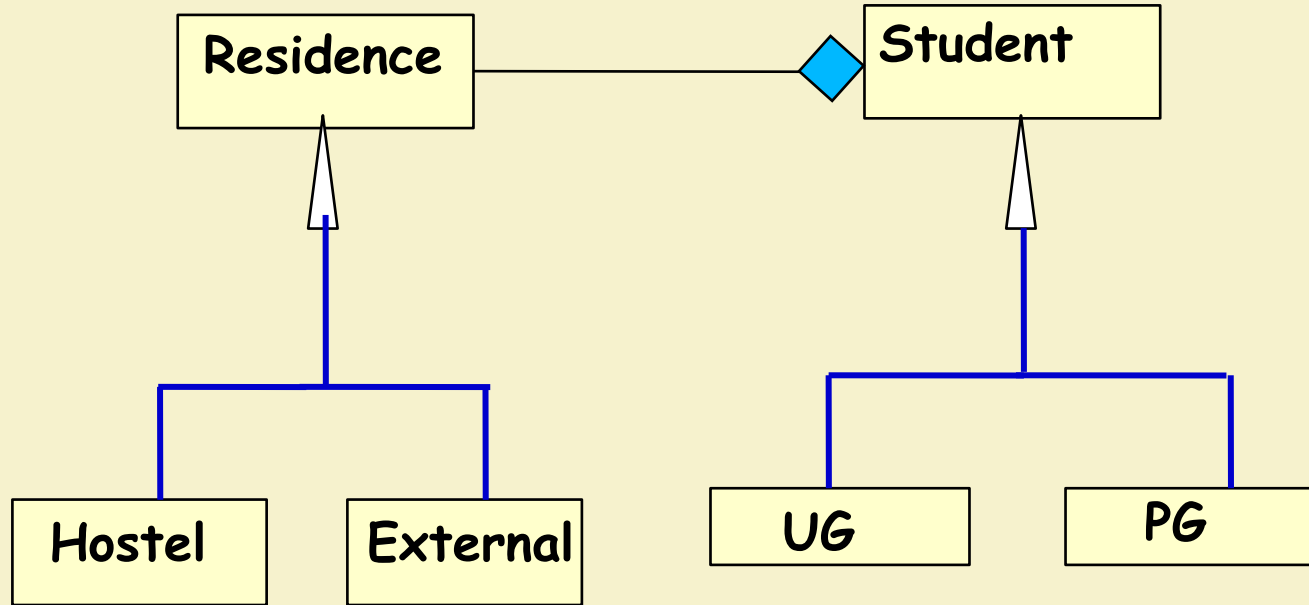


Two more
classes here...

SRP: Solution



Separate Out Abstractions and Encapsulate Variability

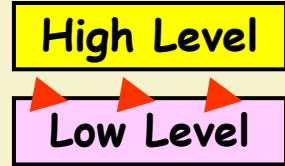


Dependency Inversion Principle (DIP)

based on an article of that title
by Robert Martin

DIP: Statement

- “Depend on Abstractions-- not on concretions”.
- High level components should not depend upon low level components.
 - Abstractions should not depend upon details...
Details should depend upon the abstractions...
- Why?
 - High-level concepts are more stable than low-level implementation.



- Whenever a certain functionality is implemented:
 - It can be divided into high level and low level parts.
- The high level parts usually represent the concept in the activity:
 - Whereas the low level parts implement the details of the high level activities.
 - **Low level activities (code) more susceptible to change...**

DIP: An Explanation

High Level

Low Level

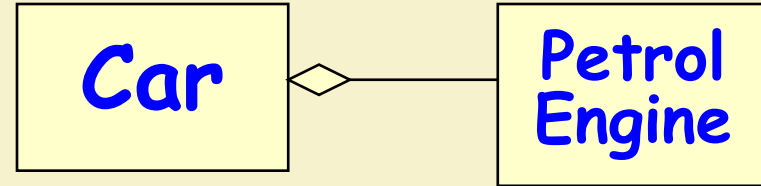
Dependency Inversion Principle

- **Higher level modules should not depend on the concrete lower level modules.**
- Should depend on abstractions
 - Interfaces or Abstract classes
- **Complex systems should be structured into layers.**
 - **But if that is not done carefully -- the top levels implicitly depend on the lower levels...**
Warranting unnecessary changes

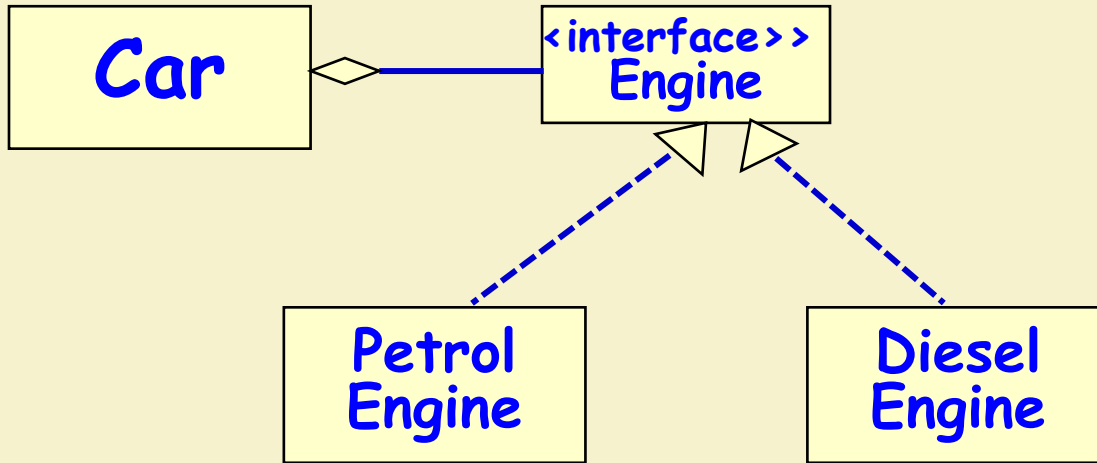
- For cost saving, you want to fit your Car with a Diesel Engine:

DIP: An Example ..

- Too bad if alterations needed to the car's internals...
- Should only change the engine...
- How to achieve?
- It should be possible to interface any standard engine ...



With DIP ...



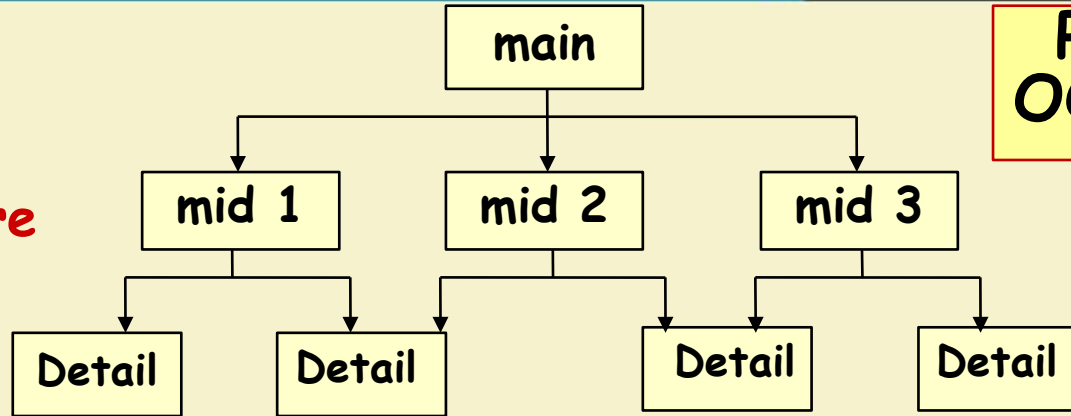
- A class must not depend on some other concrete class!
- It must depend only on an interface ...

Depend On Abstractions

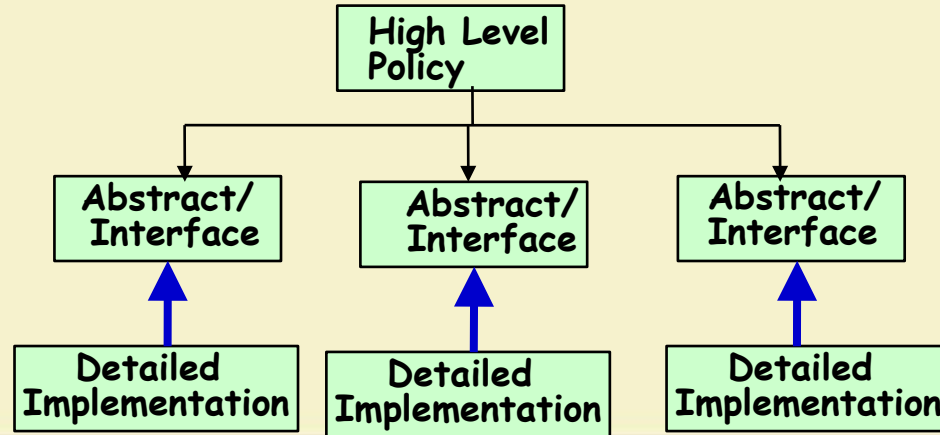
- High-level modules should not depend on low-level modules.
 - Should depend on abstractions.
- Abstractions should not depend on details.
 - **Details should depend on abstractions.**

Procedural vs. OO Architecture

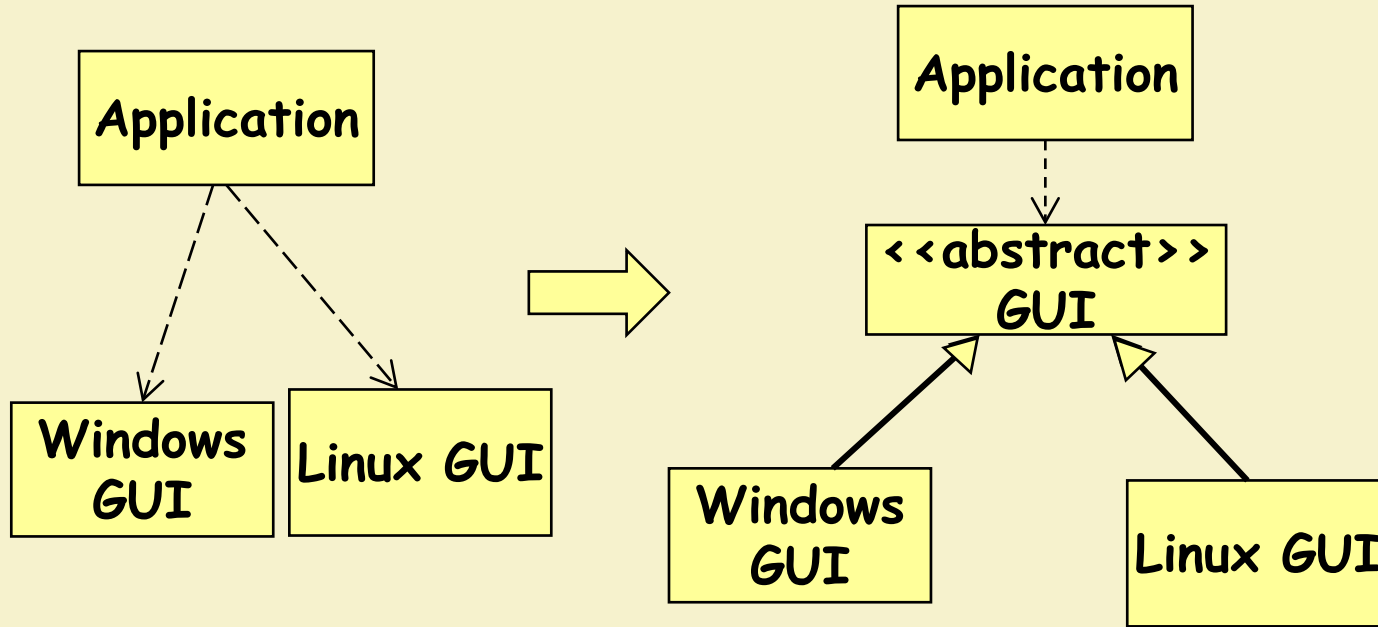
Procedural
Architecture



Object-
Oriented
Architecture



Dependency Inversion Example: 2



DIP: Summary

- Dependency Inversion Principle -- statement:
 - **Components that implement high level policy should not depend on components that implement details.**
- Instead, both kinds of components should depend on abstractions.