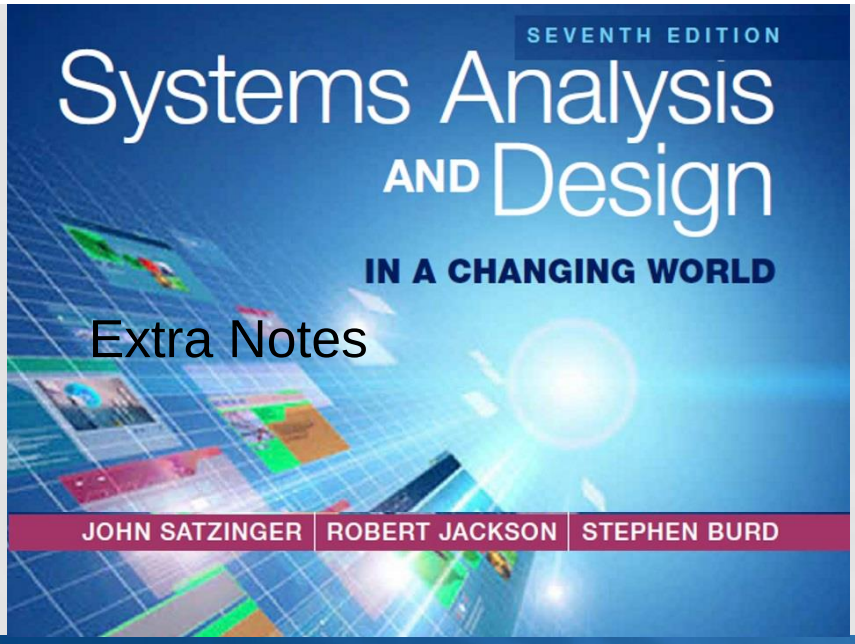


Why Inheritance?

1. Organization (during analysis):
   - Inheritance helps us with the construction of taxonomies to deal with the application domain
     - when talking the customer and application domain experts we usually find already existing taxonomies
2. Reuse (during object design):
   - Inheritance helps us to reuse models and code to deal with the solution domain
     - when talking to developers

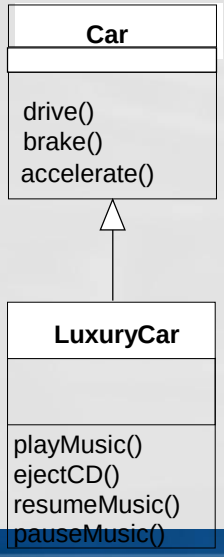
The use of Inheritance

- Inheritance is used to achieve two different goals
  - Description of Taxonomies
  - Interface Specification
- Description of Taxonomies
  - Used during *requirements analysis*
  - Activity: identify application domain objects that are hierarchically related
  - Goal: make the analysis model more understandable
- Interface Specification
  - Used during *object design*
  - Activity: identify the signatures of all identified

Inheritance can be used during Modeling as well as during Implementation

- Starting Point is always the requirements analysis phase:
  - We start with use cases
  - We identify existing objects (“class identification“)
  - We investigate the relationship between these objects; “Identification of associations“:
    - general associations
    - aggregations
    - inheritance associations.

Example of Inheritance



Superclass:

```
public class Car {
    public void drive() {…}
    public void brake() {…}
    public void accelerate() {…}
}
```

Subclass:

```
public class LuxuryCar extends Car
{
    public void playMusic() {…}
    public void ejectCD() {…}
    public void resumeMusic() {…}
    public void pauseMusic() {…}
}
```

Inheritance comes in many Flavors

Inheritance is used in four ways:

- Specialization
- Generalization
- Specification Inheritance
- Implementation Inheritance.

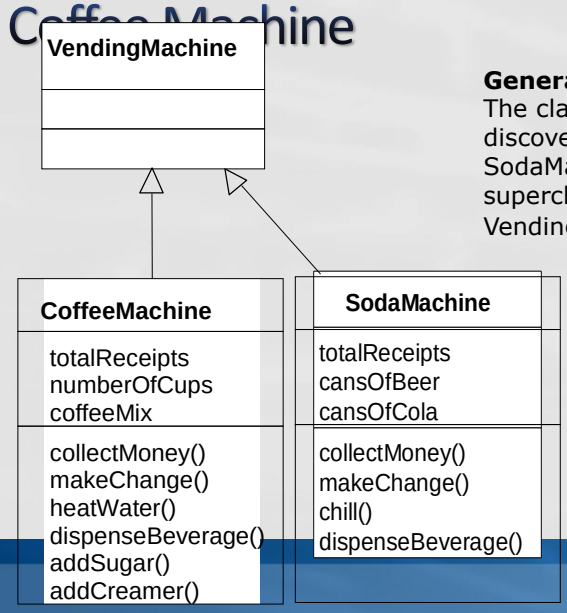
Discovering Inheritance

- To "discover" inheritance associations, we can proceed in two ways, which we call specialization and generalization
- Generalization: the discovery of an inheritance relationship between two classes, where the sub class is discovered first.
- Specialization: the discovery of an inheritance relationship between two classes, where the super class is discovered first.

Generalization

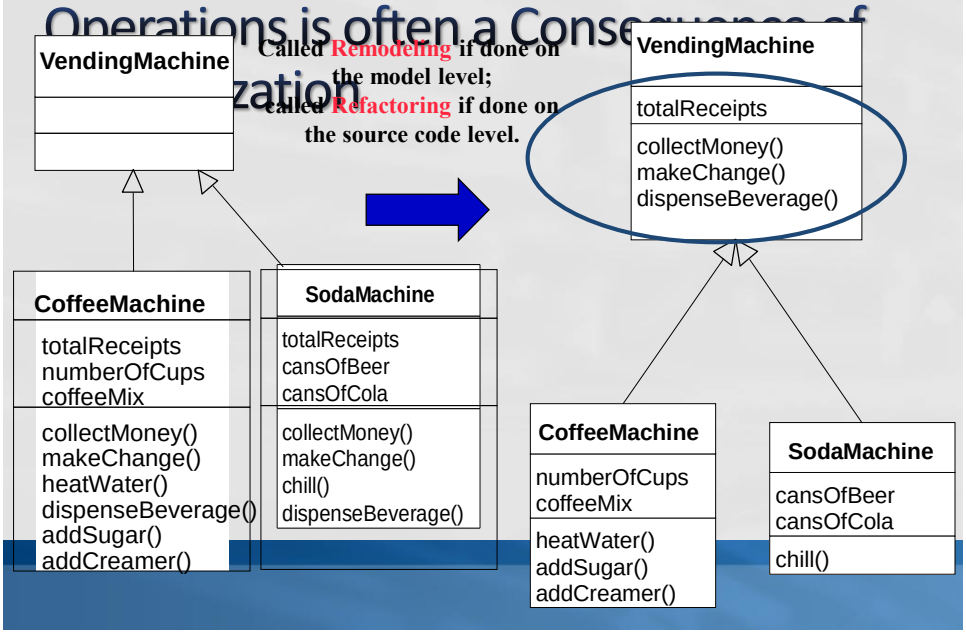
- First we find the subclass, then the super class
- This type of discovery occurs often in science and engineering:
  - **Biology**: First we find individual animals (Elefant, Lion, Tiger), then we discover that these animals have common properties (mammals).
  - **Engineering:** What are the common properties of cars and airplanes?

Generalization Example: Modeling a Coffee Machine

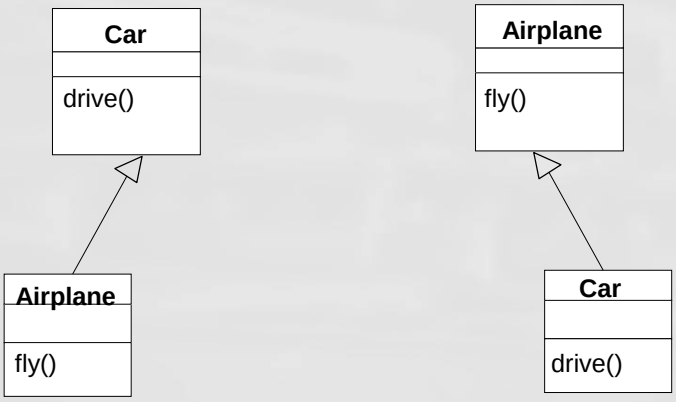


Generalization:
The class CoffeeMachine is discovered first, then the class SodaMachine, then the superclass VendingMachine

Restructuring of Attributes and Operations is often a Consequence of Generalization



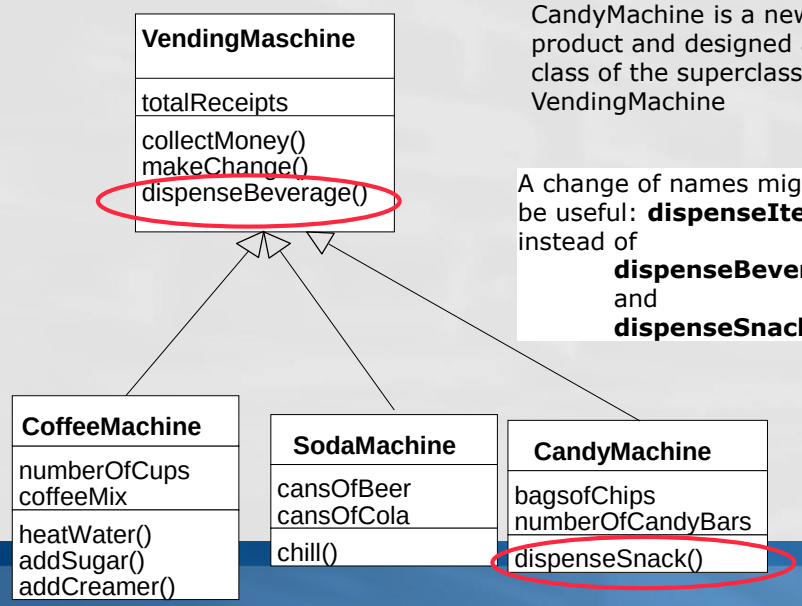
Which Taxonomy is correct for the Example in the previous Slide?



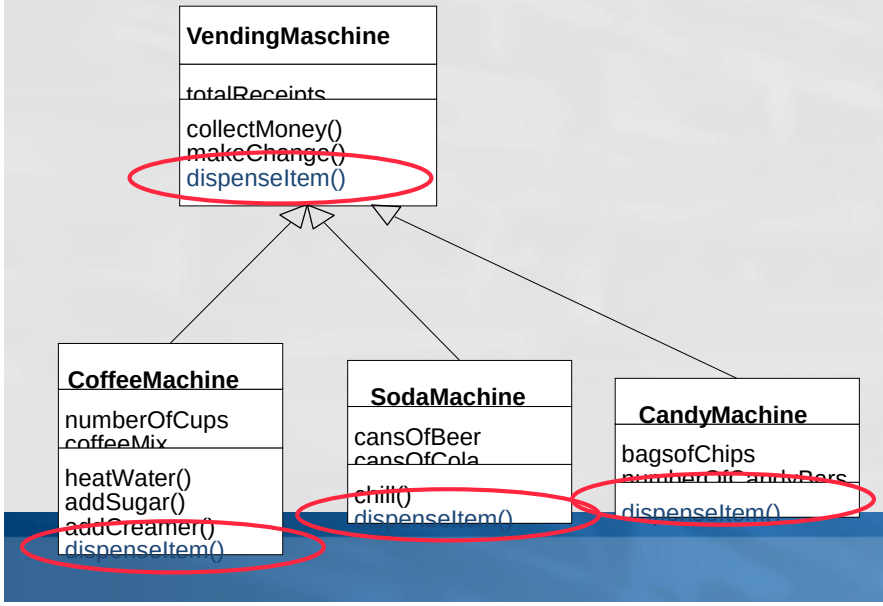
Another Example of a Specialization

CandyMachine is a new product and designed as a sub class of the superclass VendingMachine

A change of names might now be useful: **dispenseItem()** instead of **dispenseBeverage()** and **dispenseSnack()**



Example of a Specialization (2)



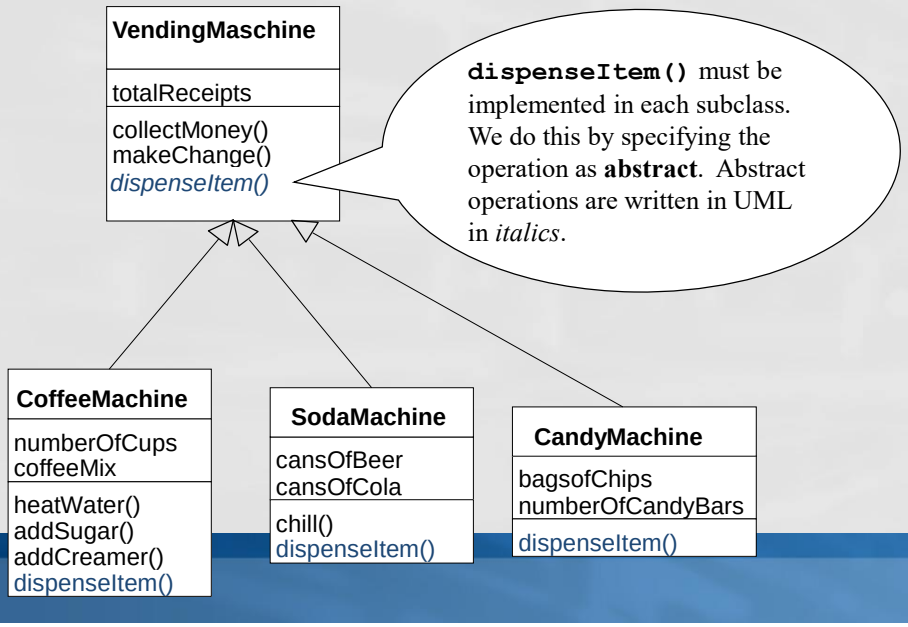
Implementation Inheritance vs. Specification Inheritance

- Implementation Inheritance: The combination of inheritance and implementation
  - The Interface of the superclass is completely inherited
  - Implementations of methods in the superclass ("Reference implementations") are inherited by any subclass
- Specification Inheritance: The combination of inheritance and specification
  - The Interface of the superclass is completely inherited
  - Implementations of the superclass (if there are any) are not inherited.

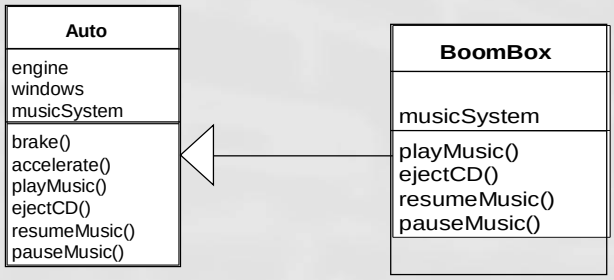
Abstract Methods and Abstract Classes

- Abstract method:
  - A method with a signature but without an implementation (also called abstract operation)
- Abstract class:
  - A class which contains at least one abstract method is called abstract class
- Interface: An abstract class which has only abstract methods
  - An interface is primarily used for the specification of a system or subsystem. The implementation is provided by a subclass or by other mechanisms.

Example of an Abstract Method



What we do to save money and time



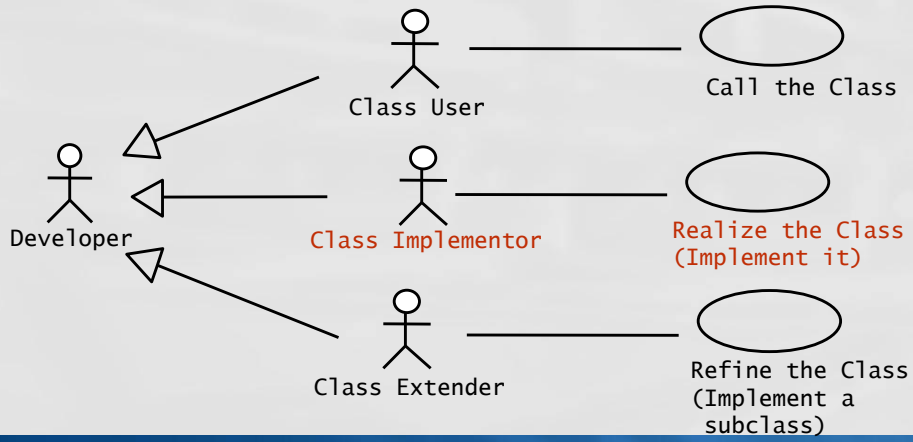
Existing Class:

```
public class Auto {
  public void drive() {…}
  public void brake() {…}
  public void accelerate() {…}
  public void playMusic() {…}
  public void ejectCD() {…}
  public void resumeMusic() {…}
  public void pauseMusic() {…}
}
```

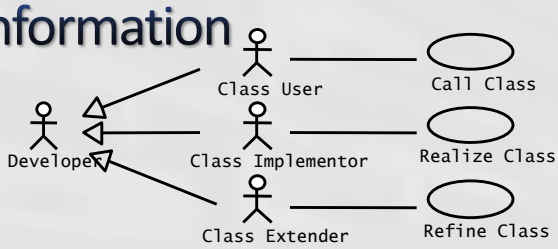
Boombox:

```
public class Boombox
extends Auto {
  public void drive() {};
  public void brake() {};
  public void accelerate()
{};
}
```

Developers play 3 different Roles during Object Design of a Class



Add Visibility Information



Class user ("Public"): +

- Public attributes/operation can be accessed by any class

Class implementor ("Private"): -

- Private attributes and operations can be accessed only by the class in which they are defined
- They cannot be accessed by subclasses or other classes

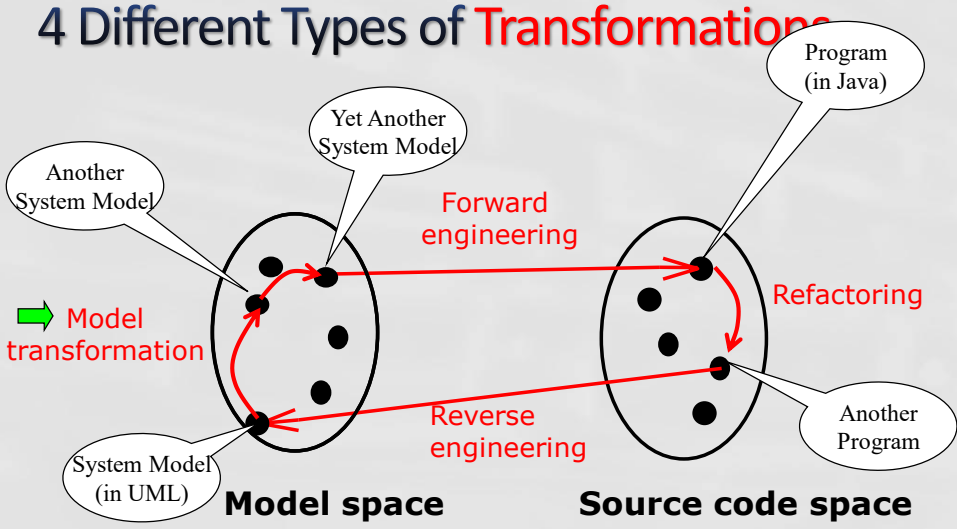
Class extender ("Protected"): #

- Protected attributes/operations can be accessed

Implementation of UML Visibility in Java

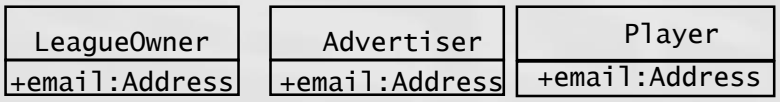

```
public class Tournament {
        private int maxNumPlayers;

    public Tournament(League l, int maxNumPlayers)
    public int getMaxNumPlayers() {…};
    public List getPlayers() {…};
    public void acceptPlayer(Player p) {…};
    public void removePlayer(Player p) {…};
    public boolean isPlayerAccepted(Player p) {…};
```

4 Different Types of Transformations

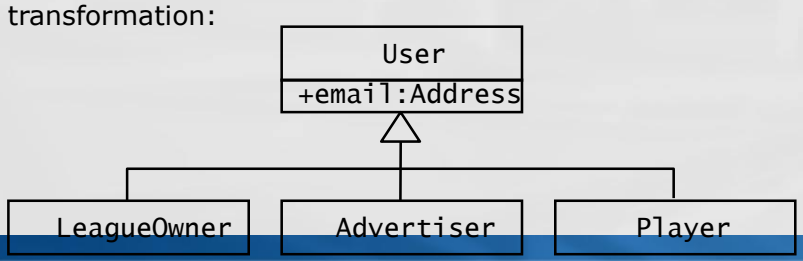


Model Transformation Example

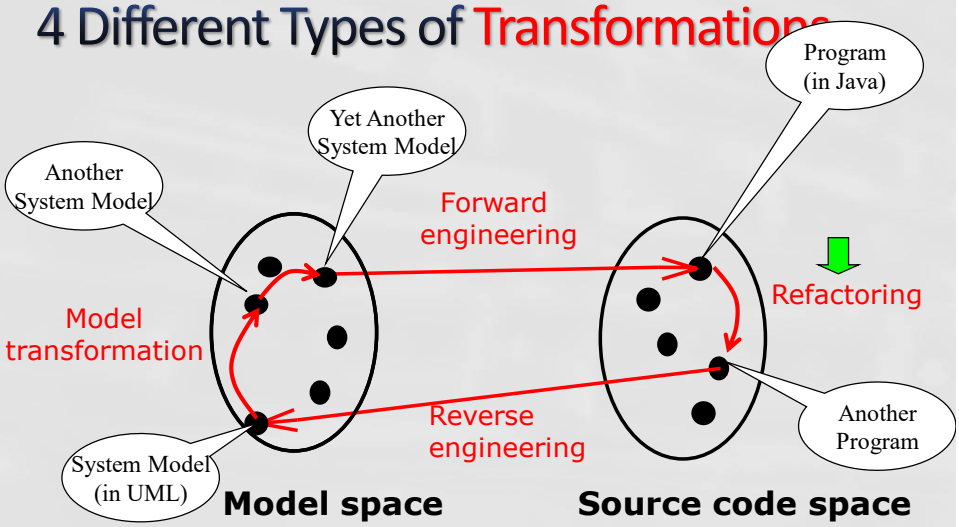
Object design model before transformation:



Object design model
after transformation:



4 Different Types of Transformations



Refactoring Example: Pull Up Field

```
public class Player {
   private String email;
   //...
}
public class LeagueOwner {
   private String eMail;
   //...
}
public class Advertiser {
   private String
   email_address;
   //...
}
```

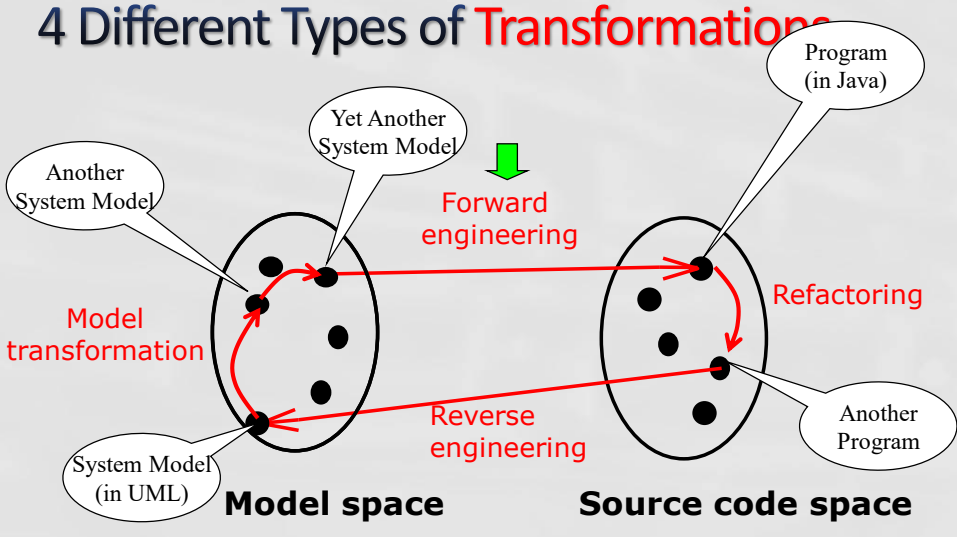
```
public class User {
   private String email;
}
public class Player extends User {
   //...
}

public class LeagueOwner extends
   User {
   //...
}
public class Advertiser extends User
   {
   //...
}
```

Refactoring Example: Pull Up Constructor Body

```
public class User {
   private String email;
}


public class Player extends User {
   public Player(String email) {
        this.email = email;
   }
}
public class LeagueOwner extends User{
   public LeagueOwner(String email) {
        this.email = email;
   }
}
public class Advertiser extendsUser{
   public Advertiser(String email) {
        this.email = email;
   }
}
```

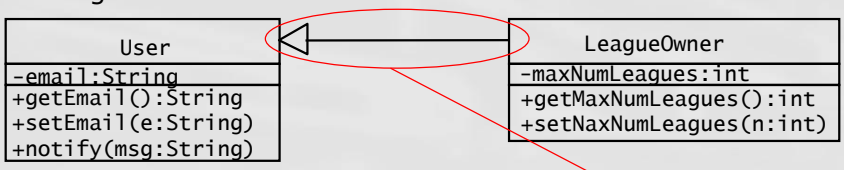
```
public class User {
   public User(String email) {
        this.email = email;
   }
}

public class Player extends User {
        public Player(String email)
{
                super(email);
        }
}
public class LeagueOwner extends
User {
        public LeagueOwner(String
email) {
                super(email);
        }
}
public class Advertiser extends Use
{
        public Advertiser(String
email) {
                super(email);
        }
}
```

4 Different Types of Transformations



Forward Engineering Example

Object design model before transformation:



Source code after transformation:

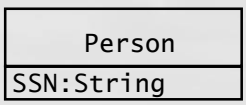
```
public class User {
    private String email;
    public String getEmail() {
          return email;
    }
    public void setEmail(String value){
          email = value;
    }
    public void notify(String msg) {
          // ....
    }
}
```

```
public class LeagueOwner extends User {
    private int maxNumLeagues;
    public int getMaxNumLeagues() {
          return maxNumLeagues;
    }
    public void setMaxNumLeagues
                (int value) {
          maxNumLeagues = value;
    }
}
```

Collapsing Objects

Object design model before transformation:

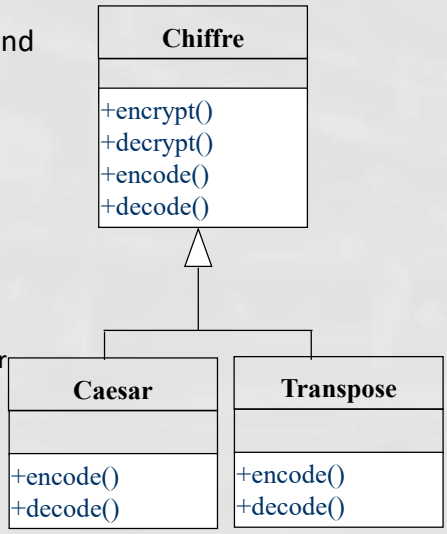
Object design model after transformation:



Turning an object into an attribute of another object is usually done, if the object does not have any interesting dynamic behavior (only get and set operations).

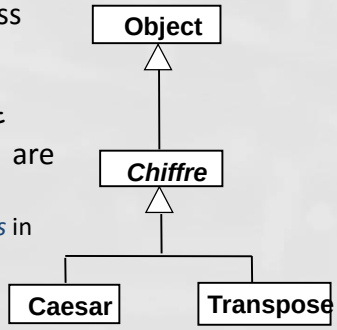
Object Design of Chiffre

- We define a super class **`Chiffre`** and define subclasses for the existing existing encryption methods
- 4 public methods:
  - **`encrypt()`** encrypts a text of words
  - **`decrypt()`** deciphers a text of words
  - **`encode()`** uses a special algorithm for encryption of a single word
  - **`decode()`** uses a special algorithm for decryption of a single word.



Implementation of Chiffre in Java

- The methods **`encrypt()`** and **`decrypt()`** are the same for each subclass and can therefore be *implemented* in the superclass **`Chiffre`**
  - **`Chiffre`** is defined as subclass of **`Object`**, because we will use some methods of **`Object`**
- The methods **`encode()`** and **`decode()`** are specific for each subclass
  - We therefore define them as *abstract methods* in the super class and expect that they are *implemented* in the respective subclasses.



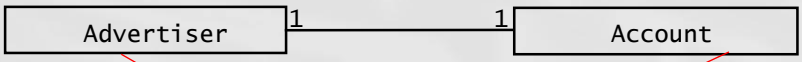
Exercise: Write the corresponding Java Code!

Mapping Associations

1. Unidirectional one-to-one association
2. Bidirectional one-to-one association
3. Bidirectional one-to-many association
4. Bidirectional many-to-many association
5. Bidirectional qualified association.

Unidirectional one-to-one association

Object design model before transformation:

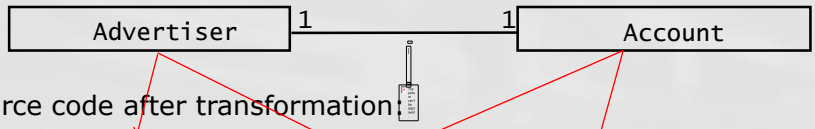


Source code after transformation:

```
public class Advertiser {
      private Account account;
      public Advertiser() {
            account = new Account();
      }

}
```

Bidirectional one-to-one association

Object design model before transformation:



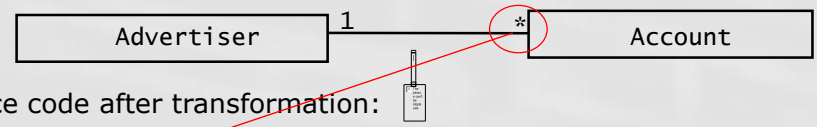
Source code after transformation:

```
public class Advertiser {
/* account is initialized
 * in the constructor and never
 * modified. */
   private Account account;
   public Advertiser() {
        this.account = account;
   }
}
```

```
public class Account {
 /* owner is initialized
  * in the constructor and
  * never modified. */
 private Advertiser owner;
 public Account() {
      this.owner = owner;
 }
 }
```

Bidirectional one-to-many association

Object design model before transformation:



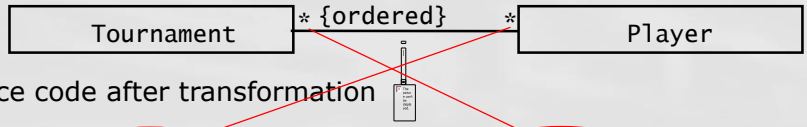
Source code after transformation:

```
public class Advertiser {
   private Set accounts;
   public Advertiser() {
        // code goes here
   }
   public void addAccount(Account a) {
        // code goes here
   }
   public void removeAccount(Account a)
   {
        // code goes here
   }
}
```

```
public class Account {
   private Advertiser owner;

public void setOwner(Advertiser
   newOwner) {
     // code goes here
     }
   }
}
```

Bidirectional many-to-many association

Object design model before transformation



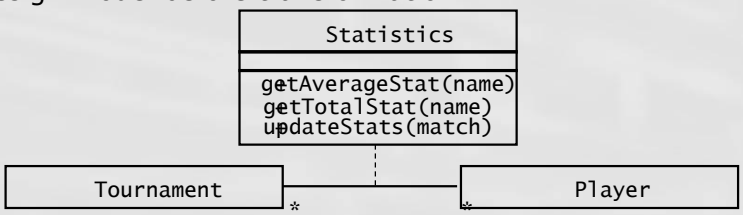
Source code after transformation

```
public class Tournament {
   private List players;

public Tournament() {
      players = new ArrayList();
   }
   public void addPlayer(Player p)
   {// code goes here
      }
   }
}
```

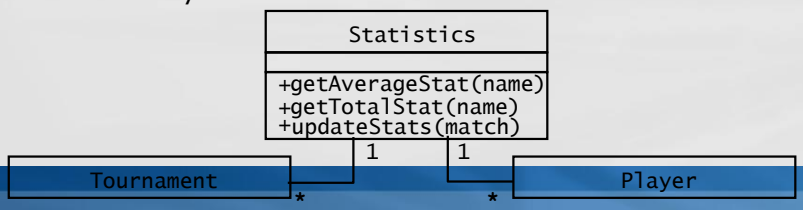
```
public class Player {
   private List tournaments;

public Player() {
      tournaments = new ArrayList();
   }
   public void // code goes here
   }
}
```

Transformation of an Association Class

Object design model before transformation

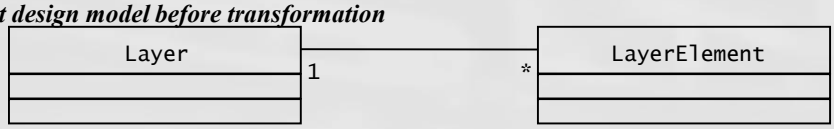


Object design model after transformation:
1 class and 2 binary associations

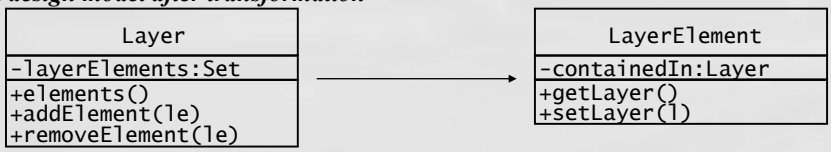


1-to-Many Association

Object design model before transformation



Object design model after transformation



Qualification

Object design model before transformation