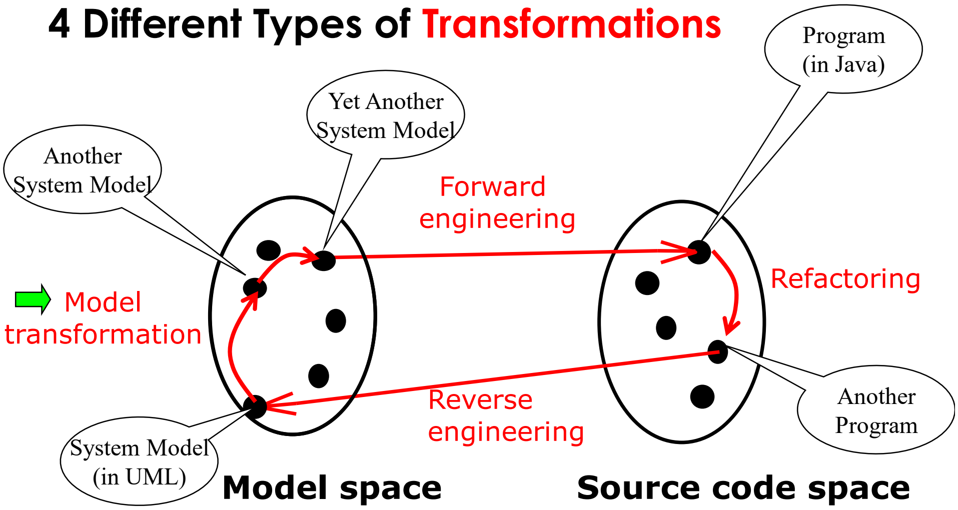
4 Different Types of Transformations



Program (in Java)

Yet Another System Model

Another System Model

Forward engineering

Refactoring

Model transformation

Another Program

Reverse engineering

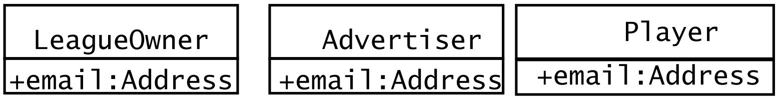
System Model (in UML)

Model space

Source code space

Model Transformation Example

Object design model before transformation:



LeagueOwner

+email:Address

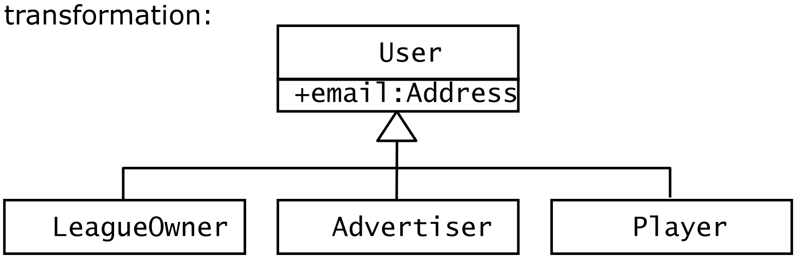
Advertiser

+email:Address

Player

+email:Address

Object design model after transformation:



User

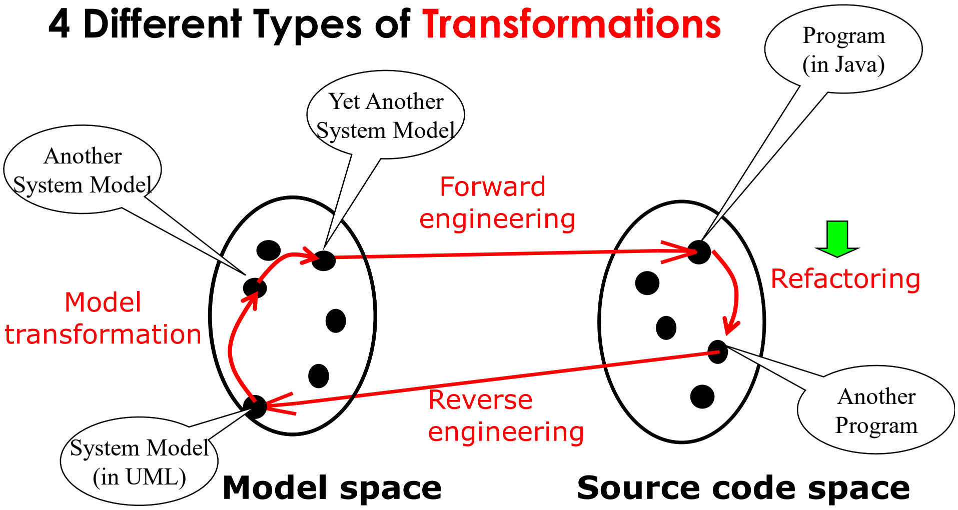
+email:Address

LeagueOwner

Advertiser

Player

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;
  //...
}
```

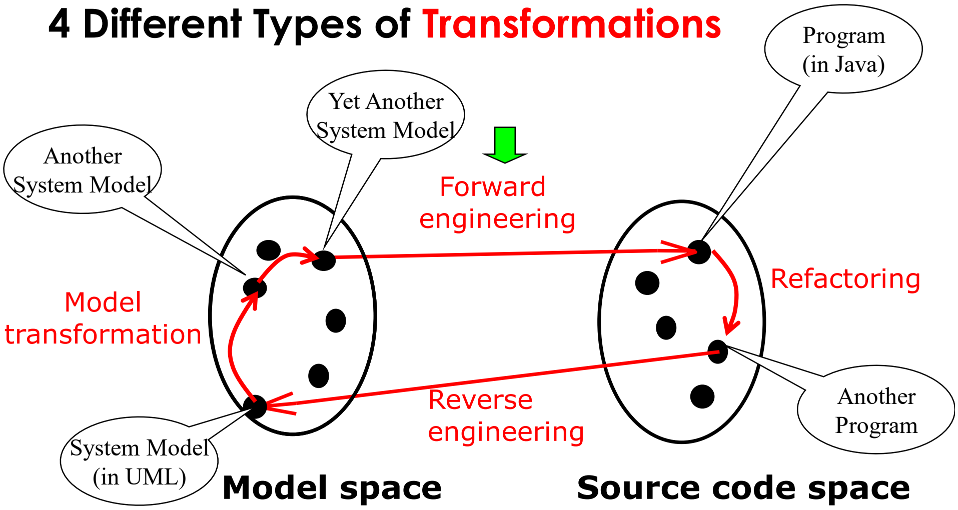
```
public class User {
  private String email;
}
public class Player extends User {
  //...
}
public class LeagueOwner extends
  User {
  //...
}
public class Advertiser extends
  User {
  //...
}
```

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;
   }
}
```

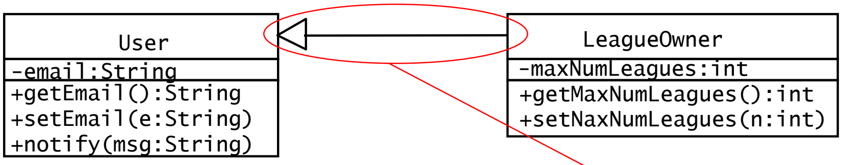
```
public class User {
   public User(String email) {
       this.email = email;
   }
}
public class Player extends User {
         public Player(String email)
{
                 super(email);
         }
}
public class LeagueOwner extends
User {
         public LeagueOwner(String
email) {
         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



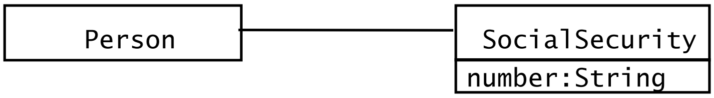
Source code after transformation:

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

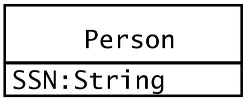
```
public class LeagueOwner extends User {
   private int maxNumLeagues;
   public int getMaxNumLeagues() {
         return maxNumLeagues;
   }
   public void setMaxNumLeagues
                  (int value) {
         maxNumLeagues = value;
   }
}
```

Collapsing Objects

Object design model before transformation:



Object design model after transformation:

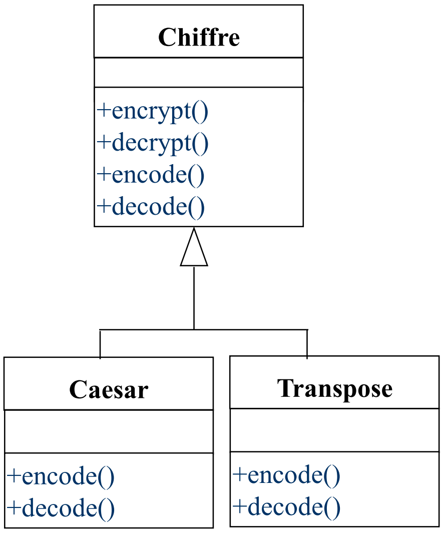


Turning an object into an attribute of another object is usually done, if the object does not have any interesting dynamic behavior (only get and set operations).

Object Design of Chiffre

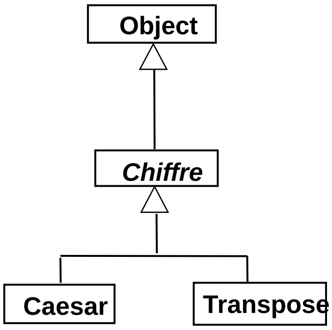
- We define a super class **Chiffre** and define subclasses for the existing existing encryption methods

- 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.



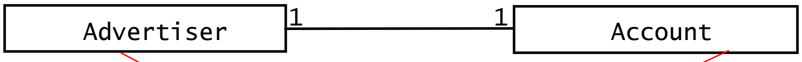
Exercise: Write the corresponding Java Code!

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:

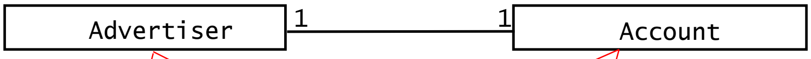


Source code after transformation:

```
public class Advertiser
{
        private Account account;

        public Advertiser() {
                account = new Account();
        }

}
```

Bidirectional one-to-one association

Object design model before transformation:



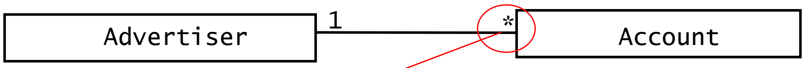
Source code after transformation:

```
public class Advertiser {
   private Account account;

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

```
public class Account {
         private Advertiser owner;

public Account() {
         owner = new Advertiser;
   }
   }
```

Bidirectional one-to-many association

Object design model before transformation:



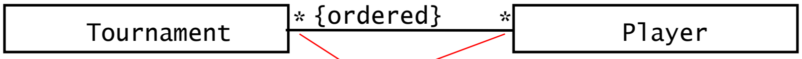
Source code after transformation:

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

   }
   }
```

```
public class Account
{
   private Advertiser owner;

    public Account() {
         owner = new Advertiser;

    }
}
```

Bidirectional many-to-many association

Object design model before transformation



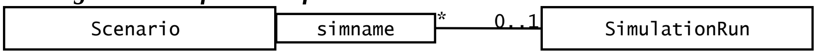
Source code after transformation

```
public class Tournament
{
   private List players;

   public Tournament() {
        players = new ArrayList();
   }

}
```

```
public class Player
{
   private List tournaments;

   public Player() {
      tournaments = new ArrayList();
   }

}
```

Qualification

Object design model before transformation


Scenario
simname
*
0..1
SimulationRun