Introduction to Software Engineering Methods

Object Design

Restructurings and
Mapping Models to Code

Literature used

- Text book

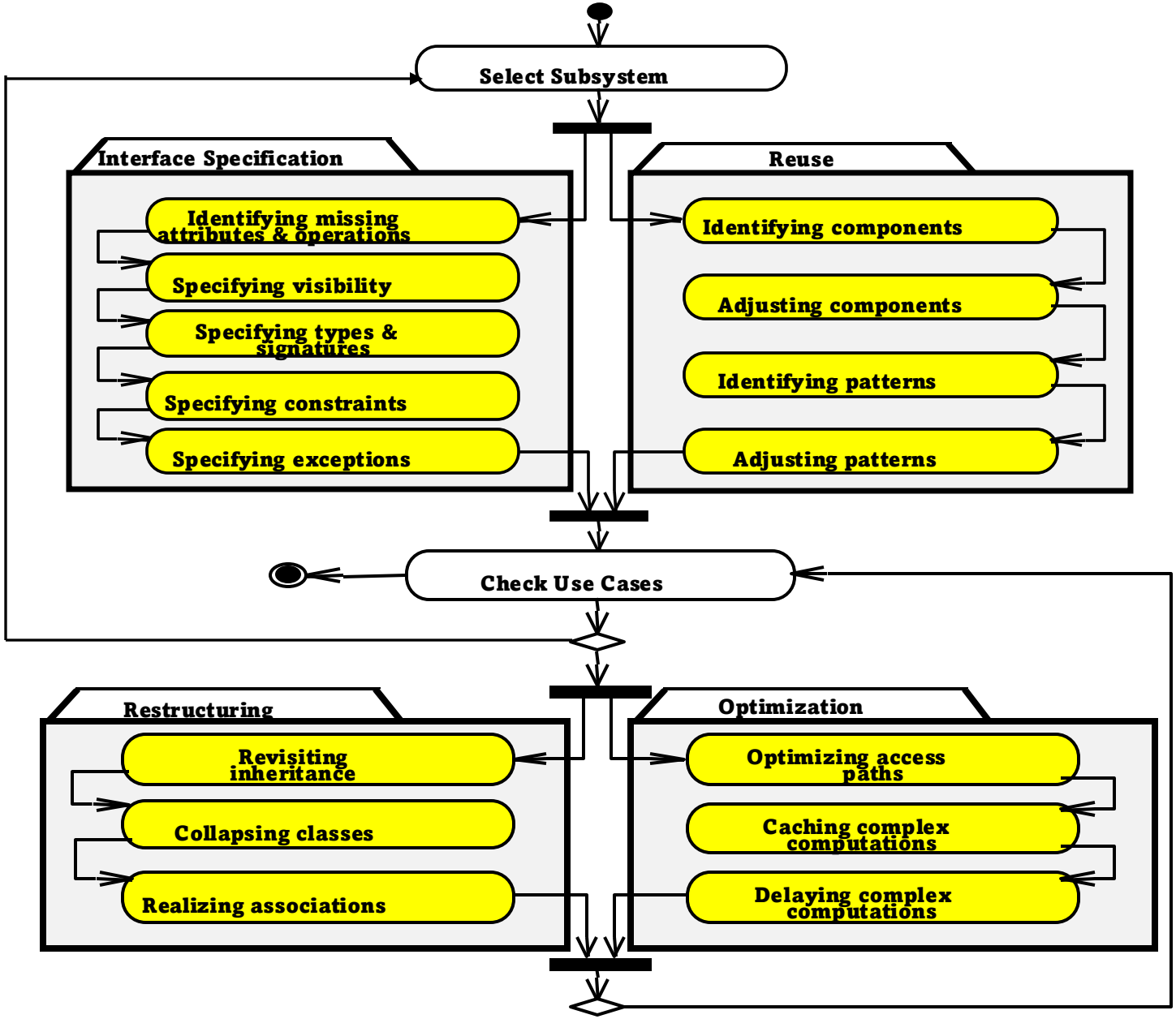
  Chapter 10

- Recommended literature:
  - Martin Fowler. ”Refactoring: Improving the Design of Existing Code”, Addison-Wesley, 2000

Introduction Content

- Restructuring, optimization and coding
- Transformations
  - Model transformations
  - Refactoring
  - Forward engineering
    - Mapping Associations
    - Mapping Contracts to Exceptions
    - Mapping Object Models to Tables

Object Design Activities

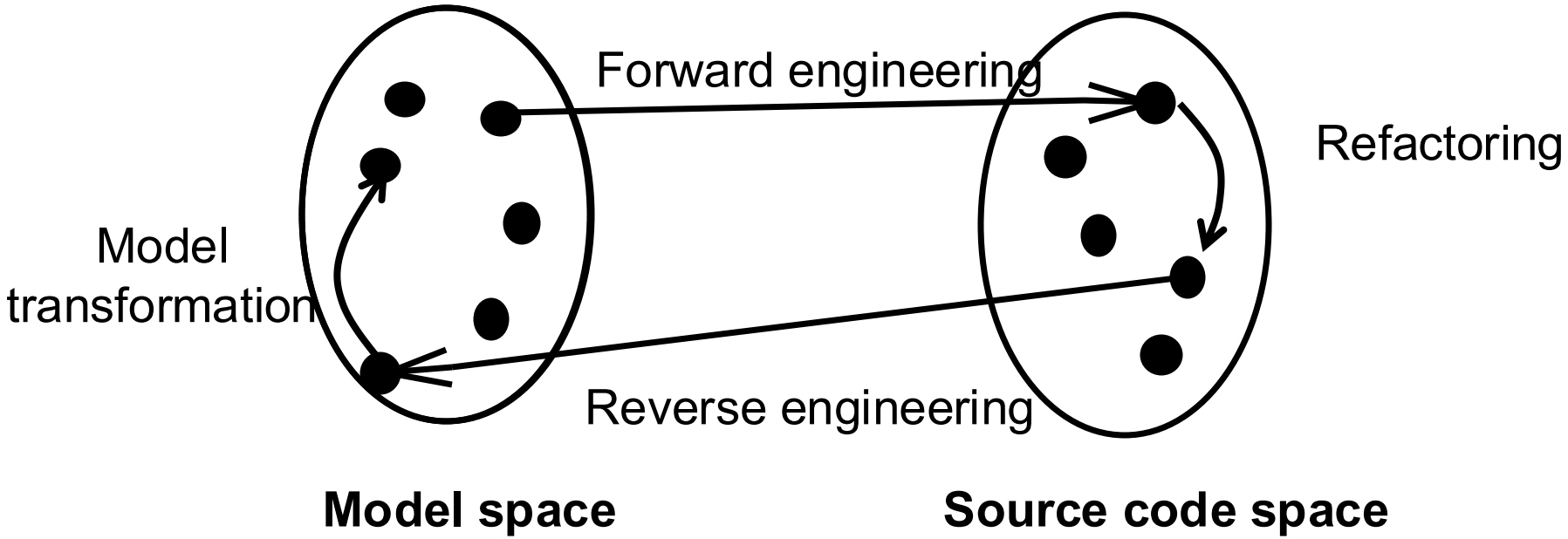


Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Restructuring, Optimization and Coding Activities

- In these activities developers perform transformations to the object model to improve its modularity and performance and to transform it to the code.
- Many of these activities are intellectually not challenging
  - However, they have a repetitive and mechanical flavor that makes them error prone.
- We describe a selection of transformations to illustrate a disciplined approach to implementation to avoid such a system degradation

Transformations



Transformation Principles

- Each transformation must address a single criterion.
- Each transformation must be local
- Each transformation must be applied in isolation to other changes
- Each transformation must be followed by a validation step.

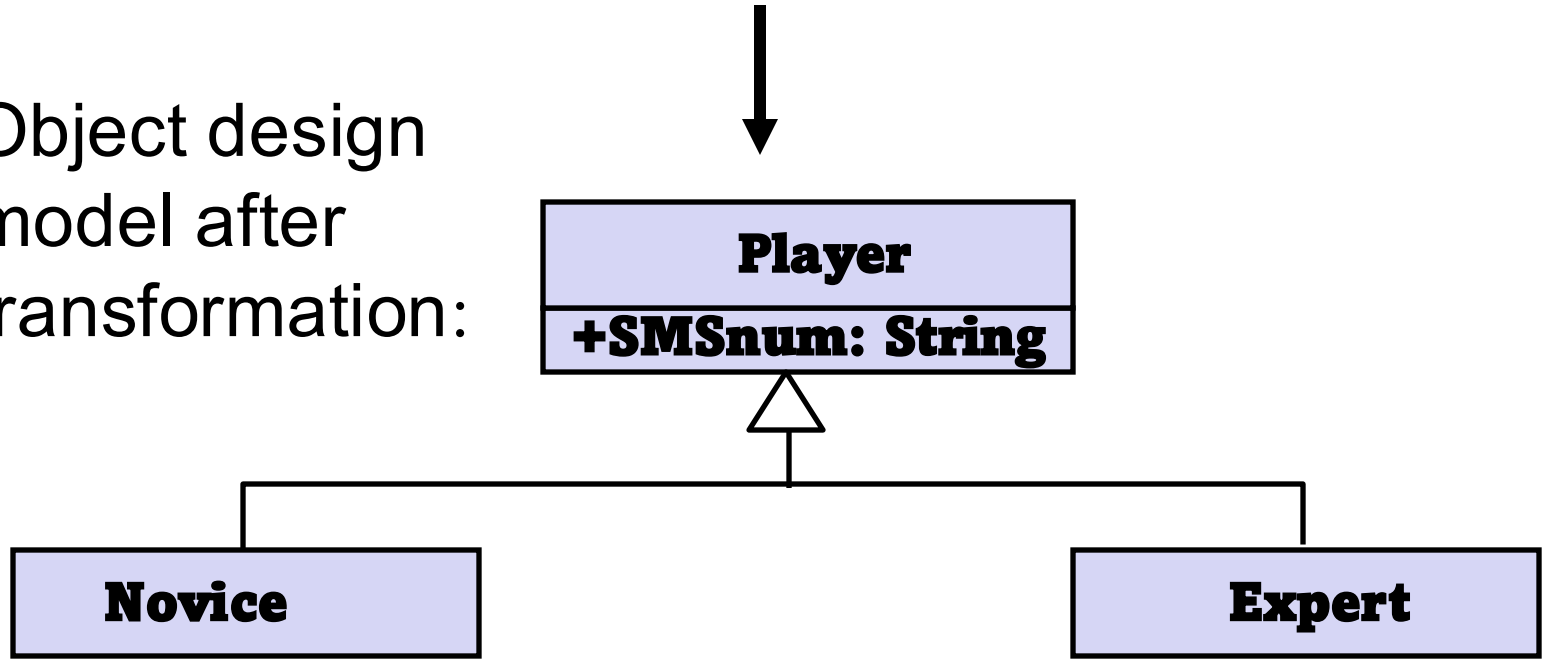
Model transformations

- Catalogue of Model Transformations : https://pdfs.semanticscholar.org/de8b/6a714d5304c82726ac643f40dae23157e146.pdf
- A Catalogue of Refactorings for Model-to-Model Transformations : http://www.jot.fm/issues/issue_2012_08/article2.pdf
- Model Transformation Design Patterns: https://nms.kcl.ac.uk/kevin.lano/mtdp/

Model Transformation Example

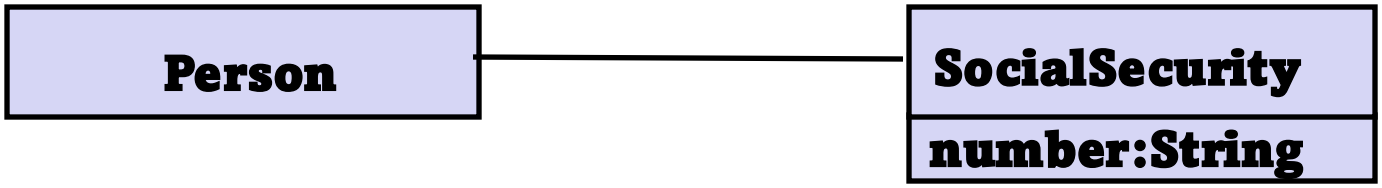
Object design model before transformation

Object design model after transformation:

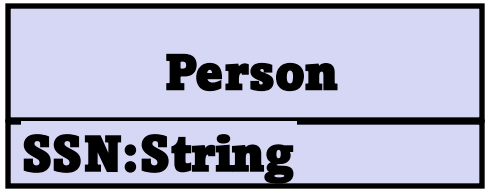


Collapsing an object without interesting behavior

Object design model before transformation



Object design model after transformation

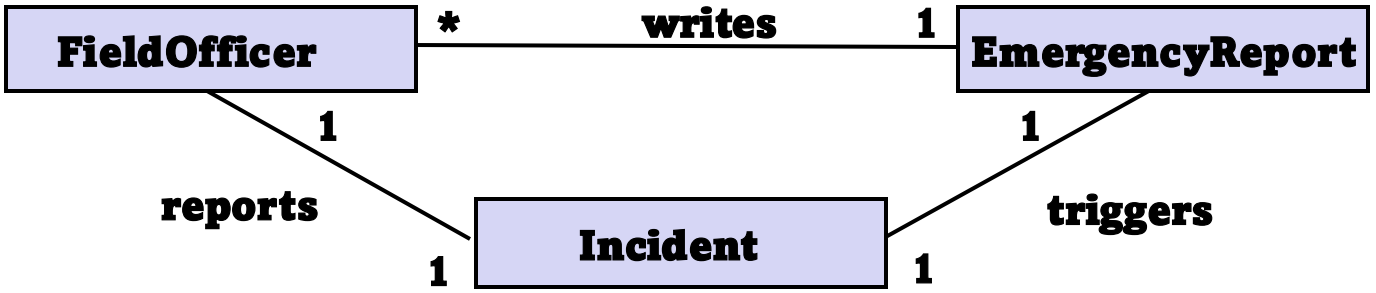


The equivalent to the model transformation (collapsing object) is Inline Class refactoring [Fowler, 2000]:

1. Declare the public fields and methods of the source class (e.g., Social Security) in the absorbing class (e.g., Person).
2. Change all references to the source class to the absorbing class.
3. Change the name of the source class to another name, so that the compiler catches any dangling references.
4. Compile and test.
5. Delete the source class.

Common Sources of Inefficiency (Optimizing access paths)

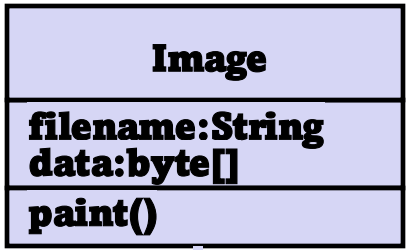
- Repeated association traversals



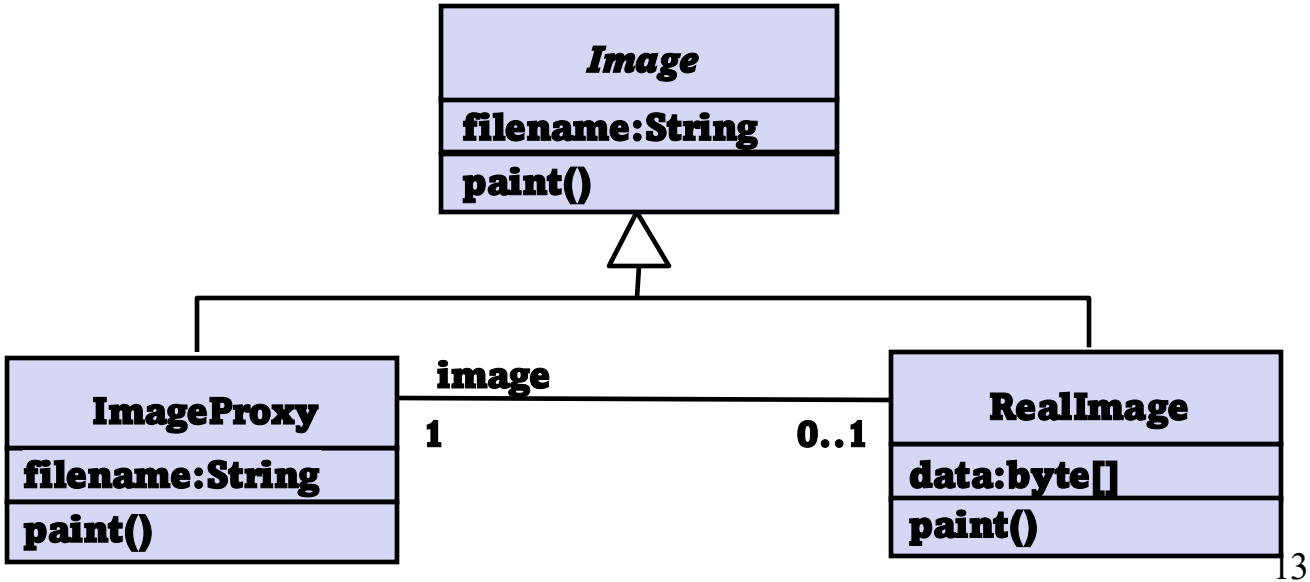
- "Many" associations (reducing)
  - qualifiers
- Misplaced attributes

Delaying expensive computations

Object design model before transformation



Object design model after transformation



Refactoring

- Improving the design of existing code without affecting its external behavior
- Before you start refactoring make sure that you have a solid set of tests

Why do we need refactoring?

- It improves the design of software
- It makes software easier to understand
- It helps to find bugs
- It helps you develop code more quickly

When should we apply refactoring?

- Refactor when you add a new function
- When you need to fix a bug
- When you do a code review

- You should not refactor
  - When rewriting from scratch
  - When you are close to deadline

What do we need for refactoring?

- Your original code
- Unit tests (to assure us we haven’t unwittingly changed the code's external behavior)
- A way to identify things to improve
- A catalog of refactorings to apply
- A process to guide us

Where to apply refactoring (bad smells in code)

- **Duplicate code** – if you see the same code structure in more than one place
- **Long method** – the object programs that live best are those with short methods
- **Large classes** – when a class is trying to do too much it often shows up as too many instance variables
- **Long parameter list**
- **Divergent changes** – when a class is commonly changed in different ways for different reasons
- **Shortgun surgery** – similar but opposite to divergent changes.
- **Feature envy** – when a method is interested in a class other than the one it actually is in
- **Data clumps** – bunches of data that hang around together in different places
- **Primitive obsession** – usage of primitive types instead of objects

Adopted from Flower

Where to apply refactoring (bad smells in code) continue

- **Parallel inheritance hierarchies** – every time you make a subclass of one class you have to make a subclass of another class
- **Lazy class** – classes that are not used enough to repay their development costs
- **Speculative generality** – consider many special cases in class or method
- **Temporary field** – you can see an object in which an instance variable is set only in certain circumstances
- **Message chains** - a long line of **getThis** methods
- **Middle man** – you look at class inheritance and found that half of methods are delegated to another class
- **Inappropriate intimacy** – some classes become far more intimate and spend too much time delving in each other private parts
- **Refused request** – some classes may not need (or even refuse) to use inherited elements
- **...**

Adopted from Flower

Composing methods

- Extract method – turn the code fragment into a method whose name explains its purpose

```
void printOwing(double amount){
    printBanner();
    …
    //print details
    System.out.println(“name” + _name);
    System.out.println(“amount” + amount);
    }
```

```
void printOwing(double amount){
    printBanner();
    printDetails(amount);
    }
void printDetails(double amount){
    System.out.println(“name” + _name);
    System.out.println(“amount” + amount);
    }
```


Adopted from Flower

Composing methods

- Inline method – put the method's body into a body of the caller and remove the method

```
int getRating(){
    return(moreThanFiveLateDeliverables()) ? 2:1);
    }
boolean moreThanFiveLateDeliverables (){
    return _numberOfLateDeliverables > 5;
}
```

```
int getRating(){){
    return (_numberOfLateDeliverables > 5) ? 2:1;
}
```

- Inline temp – replace all references to the temp with its expression

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```


Adopted from Flower

Composing methods

- Replace temp with query – extract the expression into method and replace all references to the temp with this expression.

```
double basePrice = _quantity*_itemPrice;
if (basePrice > 1000)
    return basePrice*0.95;
else
     return basePrice*0.98;
```

```
if (basePrice() > 1000)
    return basePrice()*0.95;
else
     return basePrice()*0.98;
. . .
double basePrice(){
    return _quantity*_itemPrice;
}
```

- Split temporary variable – make a separate temp for each assignment

```
double temp = 2*(_height+_width);
System.out.println(temp);
temp =_height*_width;
System.out.println(temp);
```

```
final double perimeter = 2*(_height+_width);
System.out.println(perimeter)
final double area =_height*_width;;
System.out.println(area);
```


Adopted from Flower

Organizing data

- Self encapsulate field – create getting and setting methods for the field and use only those to access the field

```
private int _low, _high;
boolean  includes (int arg) {
   return arg >= _low && arg <= _high;
}
```

```
private int _low, _high;
boolean  includes (int arg) {
   return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```

- Replace magic number with symbolic constant – create a constant, name it meaningfully and replace the number with it

```
double potentialEnergy(double mass, double height() {
   return mass*9.81*height;
}
```

```
double potentialEnergy(double mass, double height() {
   return mass*GRAVITATIONAL_CONSTANT*height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```


Adopted from Flower

Simplifying conditional expressions

- Decompose conditional – extract methods from condition, then and else parts

```
if  (date.before (SUMMER_START)|| date.after(SUMMER_END))
   charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

```
if  (notSummer(date))
   charge = winterCharge(quantity);
else charge = summerCharge(quantity);
```

- Consolidate duplicate conditional fragments – move it outside of the expression

```
if  (isSpecialDeal()) {
   total = price*0.95;
   send();
}
else {
   total = price;
   send();
}
```

```
if  (isSpecialDeal())
   total = price*0.95;
else
   total = price;
send();
```


Adopted from Flower

Making method calls simpler

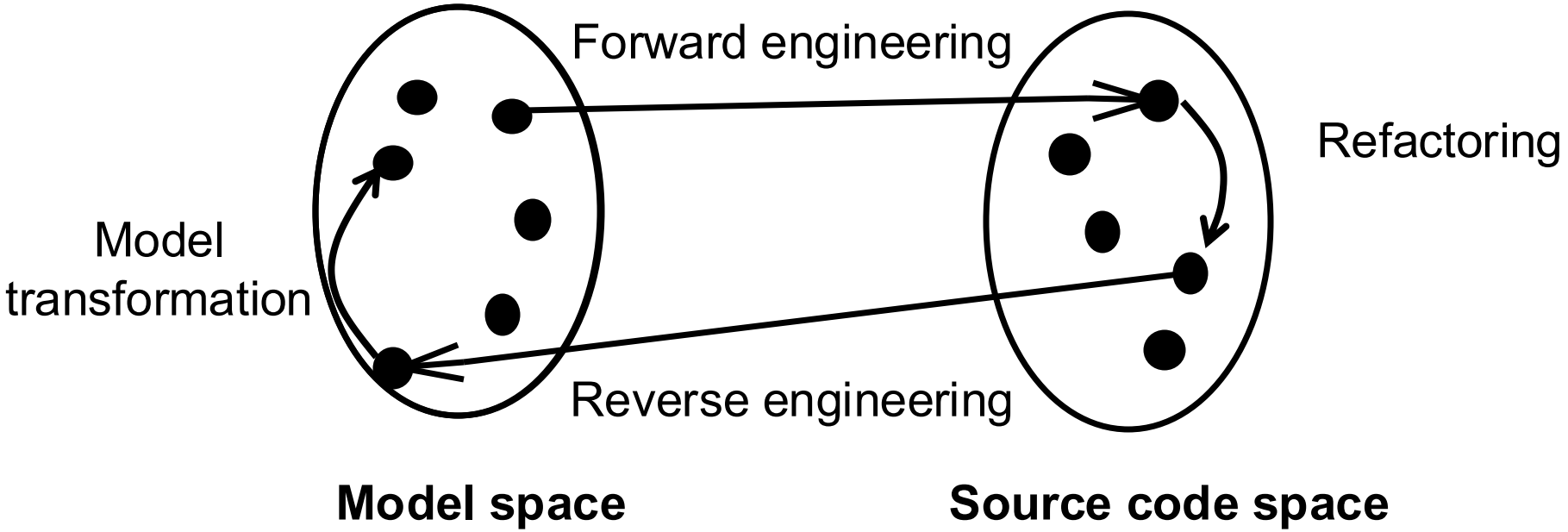
- Rename method – when name of the method doesn't reveal its purpose – change the name

```
getinvcrlim()  ⟶  getInvoiceableCreditLimit()
```

- Preserve whole object
  - if you are getting several values from an object and passing them as parameters – send the whole object instead

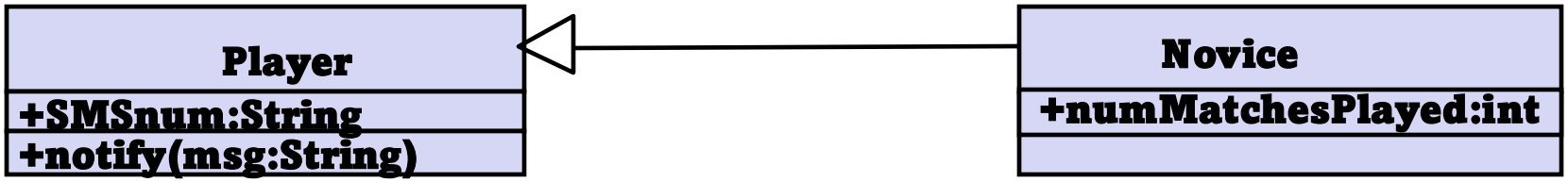
Adopted from Flower

Model transformations



Forward Engineering Example

Object design model before transformation



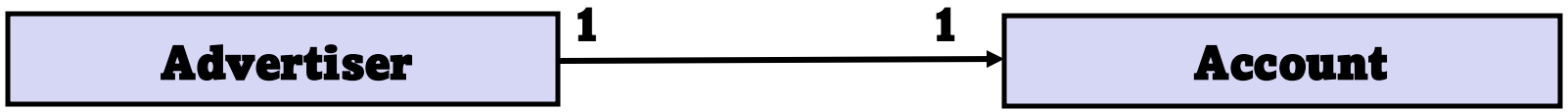
Source code after transformation

```
public class Player {
   private String SMSnum;
   public String getSMSnum() {
         return SMSnum;
   }
   public void setSMSnum(String value){
         SMSnum = value;
   }
   public void notify(String msg) {
         // ....
   }
   /* Other methods omitted */
}
```

```
public class Novice extends Player {
   private int numMatchesPlayed;
   public int getNumMatchesPlayed()
   {
         return numMatchesPlayed;
   }
   public void setNumMatchesPlayed
                (int value) {
          numMatchesPlayed = value;
   }
   /* Other methods omitted */
}
```

Implementation unidirectional, one-to-one association

Object design model before transformation



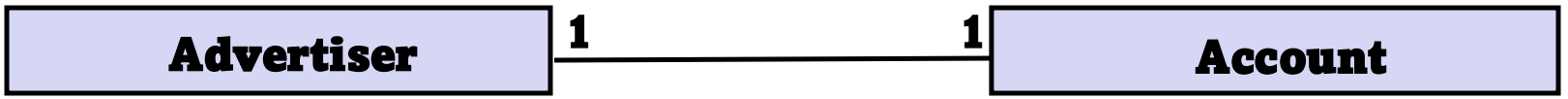
?

Source code after transformation

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

Implementing bidirectional one-to-one association

Object design model before transformation



Source code after transformation

```
public class Advertiser {
/* The account field is initialized
   * in the constructor and never
   * modified. */
   private Account account;

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

```
public class Account {
/* The owner field is initialized
   * during the constructor and
   * never modified. */
   private Advertiser owner;

   public Account(Advertiser owner){
        this.owner = owner;
   }
   public Advertiser getOwner() {
        return owner;
   }
}
```

Implementing bidirectional, one-to-many association

Object design model before transformation

Source code after transformation

```
public class Advertiser {
   private Set accounts;
   public Advertiser() {
        accounts = new HashSet();
   }
   public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
        }
   }
   public void removeAccount(Account a)
   {
        accounts.remove(a);
         . . .
   }
}
```

```
public class Account {
   private Advertiser owner;
   public void setOwner(Advertiser
          newOwner) {
     if (owner != newOwner) {
          Advertiser old = owner;
          owner = newOwner;
          if (newOwner != null)
            newOwner.addAccount(this)
          if (old != null)
            old.removeAccount(this);
     }
   }
}
```

Implementing 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)
   {
       if (!players.contains(p)) {
            players.add(p);
            p.addTournament(this);
       }
   }
}
```

```
public class Player {
   private List tournaments;
   public Player() {
       tournaments = new ArrayList();
   }
   public void addTournament(Tournament
   {
       if (!tournaments.contains(t)) {
            tournaments.add(t);
            t.addPlayer(this);
       }
   }
}
```

Exceptions as building blocks for contract violations

- Many object-oriented languages, including Java do not include built-in support for contracts.
- However, we can use their exception mechanisms as building blocks for signaling and handling contract violations
- In Java we use the try-throw-catch mechanism

The try-throw-catch Mechanism in Java

```
public class TournamentControl {
    private Tournament tournament;
    public void addPlayer(Player p) throws KnownPlayerException {
        if (tournament.isPlayerAccepted(p)) {
                throw new KnownPlayerException(p);
        }
        //... Normal addPlayer behavior
    }
}
public class TournamentForm {
    private TournamentControl control;
    private ArrayList players;
    public void processPlayerApplications() { // Go through all the players
        for (Iteration i = players.iterator(); i.hasNext();) {
            try {         // Delegate to the control object.
                control.addPlayer((Player)i.next());
            } catch (KnownPlayerException e) {
                // If an exception was caught, log it to the console
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}
```

Implementing a contract

For each operation in the contract, do the following

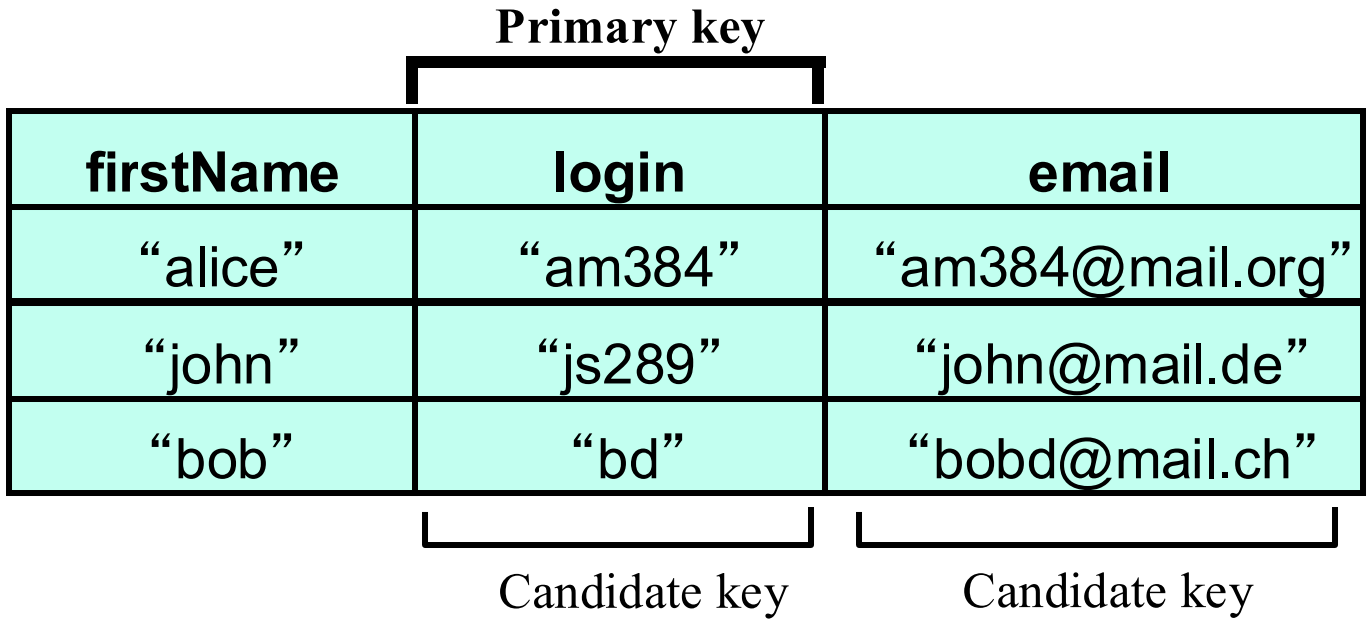
- **Check precondition**: Check the precondition before the beginning of the method with a test that raises an exception if the precondition is false.
- **Check postcondition:** Check the postcondition at the end of the method and raise an exception if the contract is violated. If more than one postcondition is not satisfied, raise an exception only for the first violation.
- **Check invariant:** Check invariants at the same time as postconditions.
- **Deal with inheritance:** Encapsulate the checking code for preconditions and postconditions into separate methods that can be called from subclasses.

Mapping an object model to a relational database

- UML mappings
  - Each *class* is mapped to a table
  - Each class *attribute* is mapped onto a column in the table
  - An *instance* of a class represents a row in the table
  - *A many-to-many association* is mapped into its own table
  - A *one-to-many association* is implemented as buried foreign key
- Methods are not mapped

Example for Primary and Foreign Keys

User table

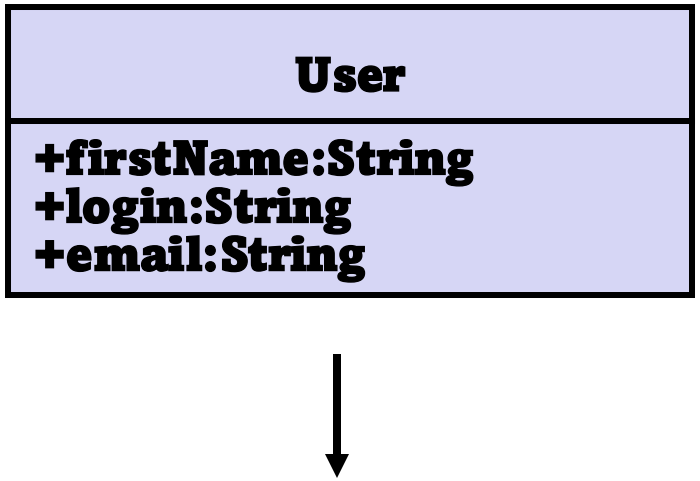


League table

name	login
“tictactoeNovice”	“am384”
“tictactoeExpert”	“am384”
“chessNovice”	“js289”

Foreign key referencing **User table**

Mapping the User class to a database table

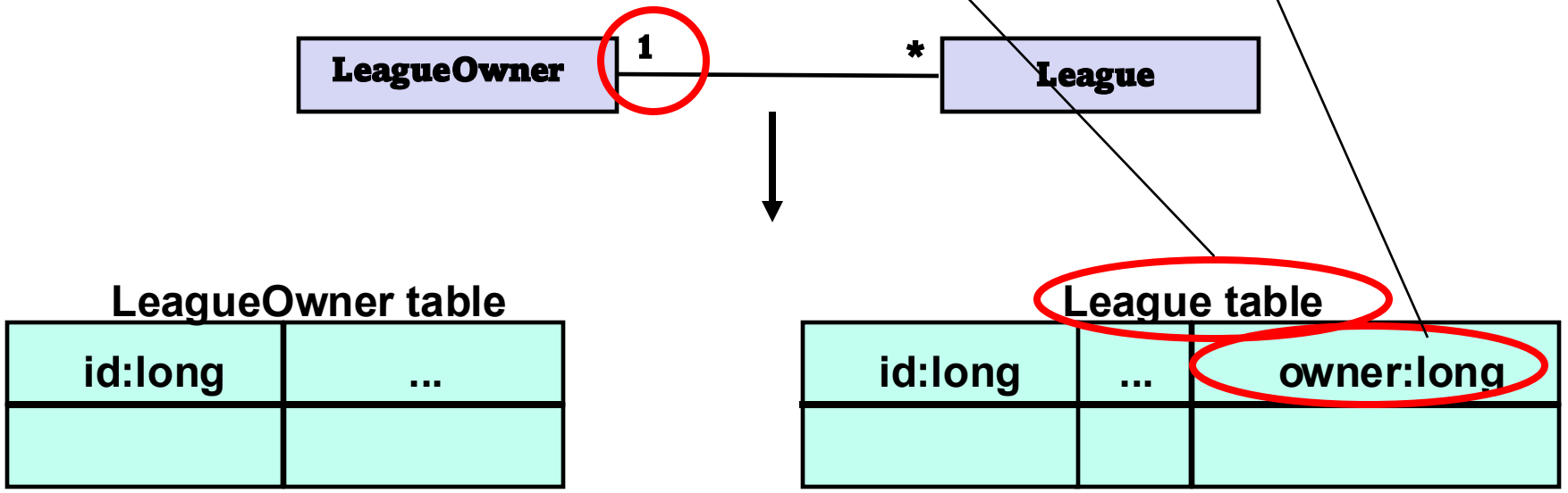


User table

id:long	firstName:text[25]	login:text[8]	email:text[32]

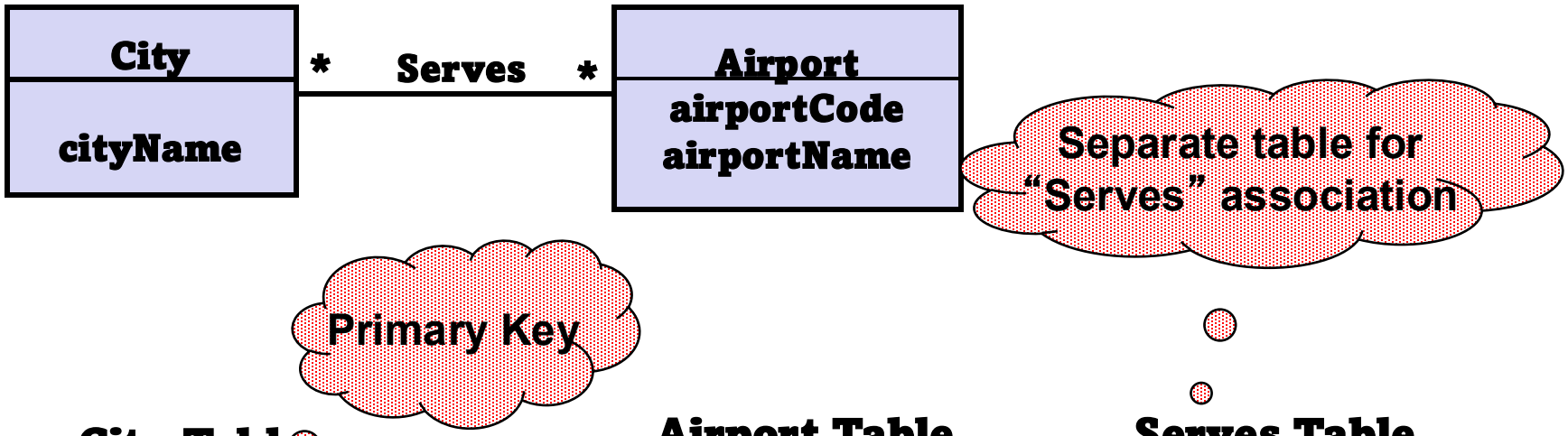
Buried Association

- Associations with multiplicity "one" can be implemented using a foreign key. Because the association vanishes in the table, we call this a buried association.
- For one-to-many associations we add the foreign key to the table representing the class on the "many" end.



Mapping Many-To-Many Associations

In this case we need a separate table for the association



City Table

cityName
Houston
Albany
Munich
Hamburg

Airport Table

airportCode	airportName
IAH	Intercontinental
HOU	Hobby
ALB	Albany County
MUC	Munich Airport
HAM	Hamburg Airport

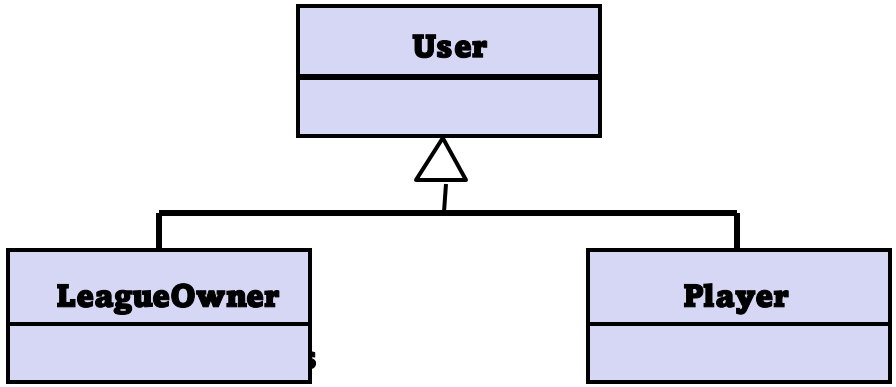
Serves Table

cityName	airportCode
Houston	IAH
Houston	HOU
Albany	ALB
Munich	MUC
Hamburg	HAM

Mapping Inheritance

- Relational databases do not support inheritance
- Three possibilities to map UML inheritance relationships to a database schema
  - With a separate table (vertical mapping)
    - The attributes of the superclass and the subclasses are mapped to different tables
  - By duplicating columns (horizontal mapping)
    - There is no table for the superclass
    - Each subclass is mapped to a table containing the attributes of the subclass and the attributes of the superclass
  - With one table(horizontal mapping)
    - There is no table for the superclass
    - All subclasses are mapped to a table containing all attributes of all subclasses and superclass

Implementing inheritance as a separate table



User table

id	name	...	role
56	zoe		LeagueOwner
79	john		Player

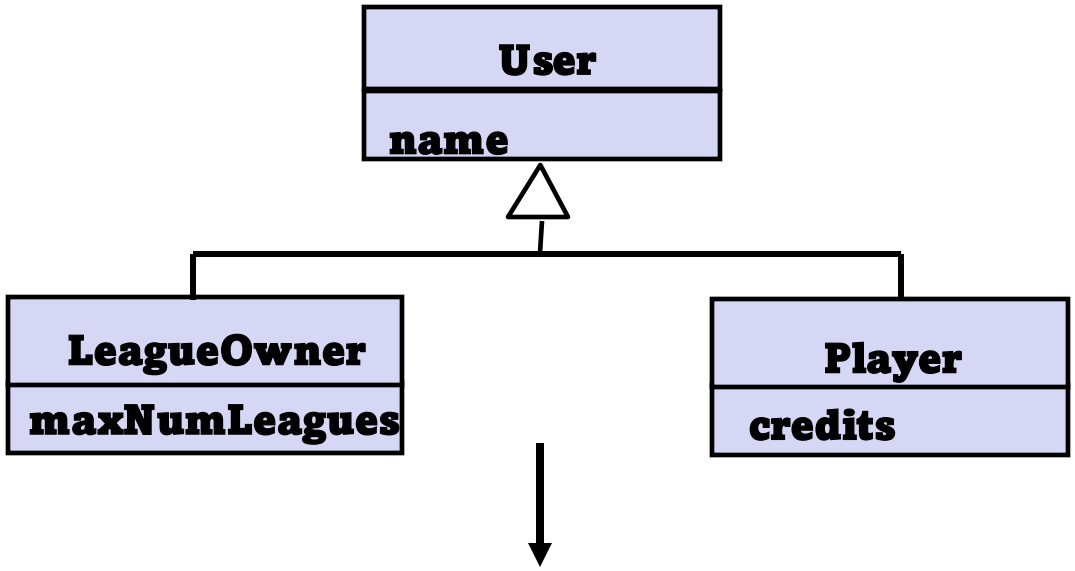
LeagueOwner table

id	maxNumLeagues	...
56	12	

Player table

id	credits	...
79	126	

Implemeting inheritance by duplicating columns



LeagueOwner table

id	name	maxNumLeagues	...
56	zoe	12	

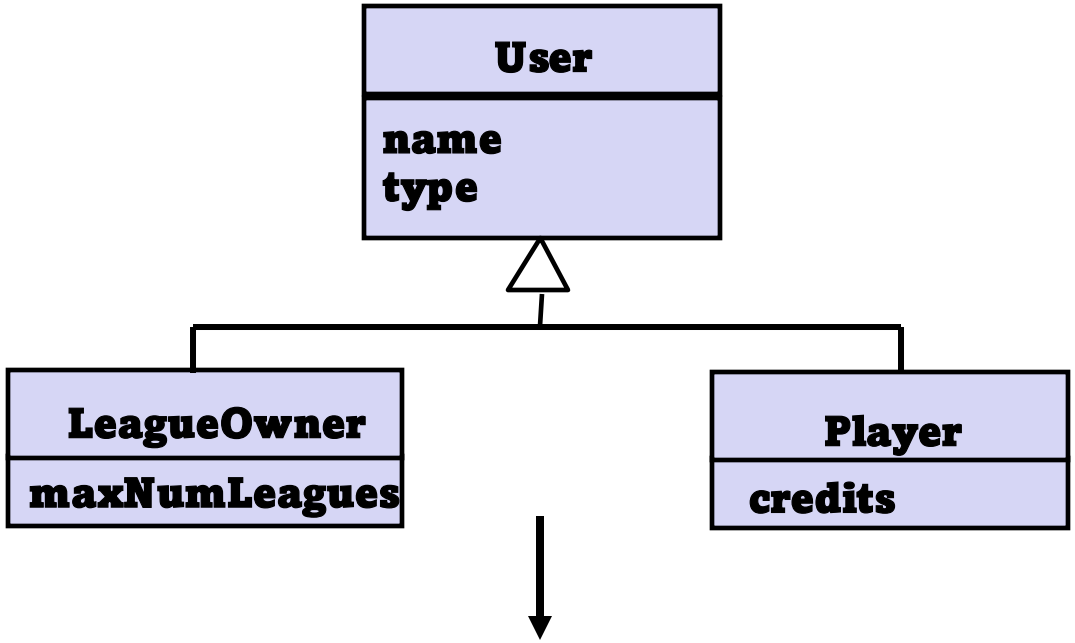
Player table

id	name	credits	...
79	john	126	

Comparison: Separate Tables vs Duplicated Columns

- The trade-off is between modifiability and response time
- Separate table mapping
  - ☺ We can add attributes to the superclass easily by adding a column to the superclass table
  - ☹ Searching for the attributes of an object requires a join operation.
- Duplicated columns
  - ☹ Modifying the database schema is more complex and error-prone
  - ☺ Individual objects are not fragmented across a number of tables, resulting in faster queries

Implementing inheritance by single columns



SingleTable

id	name	Type	maxNumLeagues	credits	Discriminator
56	zoe	12	4	Null	LeagueOwner
79	john	126	Null	115	Player

Problems of Single Table Implementation

- Pros:
  1. It is easy to implement.
  2. It doesn't need Join operation.
  3. It allows adding attributes to classes
- Cons:
  1. Big potential table size
  2. Redundancy issues
  3. Scalability can be affected when you add more columns to the table for each new subclass

Summary

- Undisciplined changes => degradation of the system model
- Four mapping concepts were considered
  - Model transformation
  - Forward engineering
  - Refactoring
  - Reverse engineering
- We reviewed model transformation, refactoring and forward engineering techniques:
  - Optimizing the class model
  - Mapping associations to collections
  - Mapping contracts to exceptions
  - Mapping class model to storage schemas

Next Lecture

- Agile methods, Extreme Programming

  http://www.extremeprogramming.org/start.html