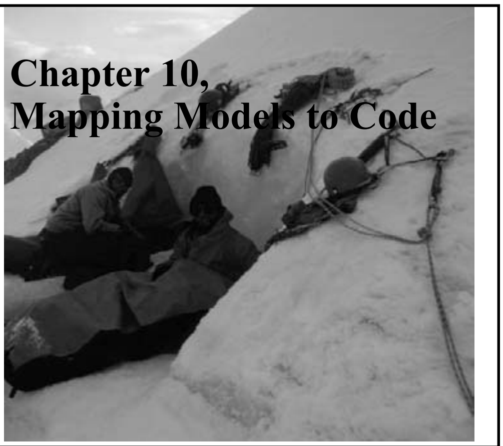
Object-Oriented Software Engineering
Using UML, Patterns, and Java



Chapter 10, Mapping Models to Code

Overview

- Object design is situated between system design and implementation. Object design is not very well understood and if not well done, leads to a bad system implementation.
- In this lecture, we describe a selection of transformations to illustrate a disciplined approach to implementation to avoid system degradation.
  1. **Operations on the object model:**
     - **Optimizations to address performance requirements**
  2. **Implementation of class model components:**
     - **Realization of associations**
     - **Realization of operation contracts**
  3. **Realizing entity objects based on selected storage strategy**
     - **Mapping the class model to a storage schema**

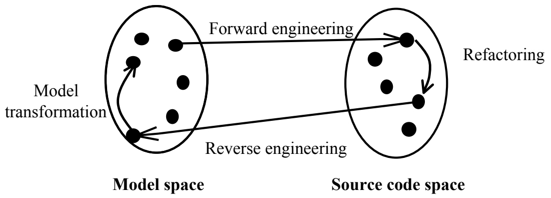
Characteristics of Object Design Activities

- Developers perform transformations to the object model to improve its modularity and performance.
- Developers transform the associations of the object model into collections of object references, because programming languages do not support the concept of association.
- If the programming language does not support contracts, the developer needs to write code for detecting and handling contract violations.
- Developers often revise the interface specification to accommodate new requirements from the client.
- All these activities are intellectually not challenging
  - **However, they have a repetitive and mechanical flavor that makes them error prone.**

State of the Art of Model-based Software Engineering

- The Vision
  - **During object design we would like to implement a system that realizes the use cases specified during requirements elicitation and system design.**
- The Reality
  - **Different developers usually handle contract violations differently.**
  - **Undocumented parameters are often added to the API to address a requirement change.**
  - **Additional attributes are usually added to the object model, but are not handled by the persistent data management system, possibly because of a miscommunication.**
  - **Many improvised code changes and workarounds that eventually yield to the degradation of the system.**

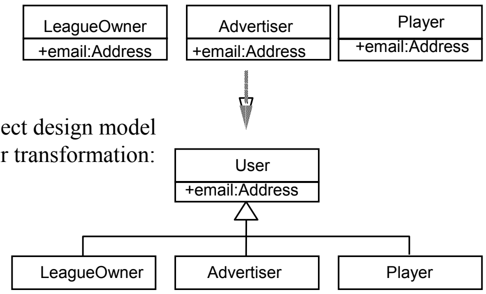
Model transformations



Model Transformation Example

Object design model before transformation

Object design model after transformation:



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 {
    //...
}
```

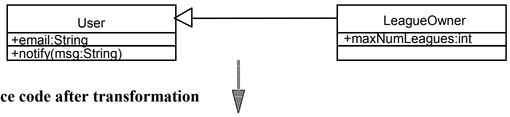
Refactoring Example: Pull Up Constructor Body

```
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 User {
    public Advertiser(String email) {
            super(email);
    }
}
```

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) {
            // ....
    }
    /* Other methods omitted */
}
```

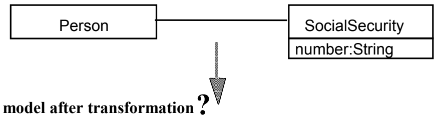
```
public class LeagueOwner extends User {
    private int maxNumLeagues;
    public int getMaxNumLeagues() {
            return maxNumLeagues;
    }
    public void setMaxNumLeagues
                    (int value) {
            maxNumLeagues = value;
    }
    /* Other methods omitted */
}
```

Other Mapping Activities

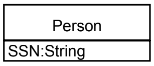
- Optimizing the Object Design Model
- Mapping Associations
- Mapping Contracts to Exceptions
- Mapping Object Models to Tables

Collapsing an object without interesting behavior

Object design model before transformation

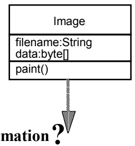


Object design model after transformation ?

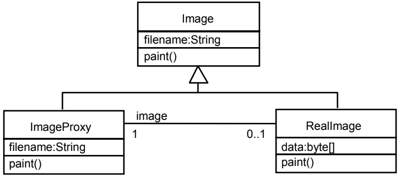


Delaying expensive computations

Object design model before transformation



Object design model after transformation ?

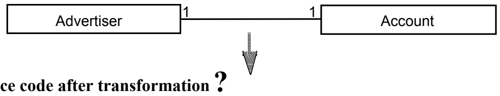


Other Mapping Activities

- ✓ Optimizing the Object Design Model
- ➢ Mapping Associations
- ♦ Mapping Contracts to Exceptions
- ♦ Mapping Object Models to Tables

Realization of a unidirectional, one-to-one association

Object design model before transformation

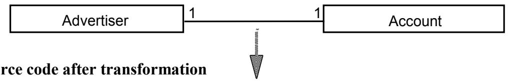


Source code after transformation ?

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

Bidirectional one-to-one association

Object design model before transformation



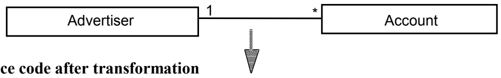
Source code after transformation

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

Bidirectional, one-to-many association

Object design model before transformation



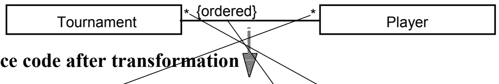
Source code after transformation

```
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);
         a.setOwner(null);
   }
}
```

```
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 (oldOwner != null)
                  old.removeAccount(this);
      }
   }
}
```

Bidirectional, many-to-many association

Object design model before transformation



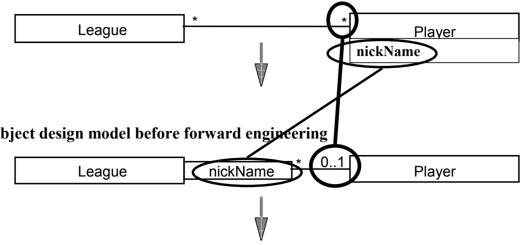
Source code after transformation

```
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 t)
   {
         if (!tournaments.contains(t)) {
          tournaments.add(t);
          t.addPlayer(this);
         }
   }
}
```

Bidirectional qualified association

Object design model before transformation



Object design model before forward engineering

Source code after forward engineering

Bidirectional qualified association (continued)

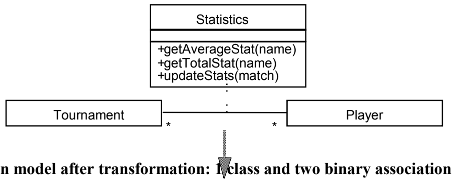
Source code after forward engineering

```
public class League {
  private Map players;

  public void addPlayer
       (String nickName, Player p) {
    if (!players.containsKey(nickName)) {
              players.put(nickName, p);
              p.addLeague(nickName, this);
        }
    }
}
```

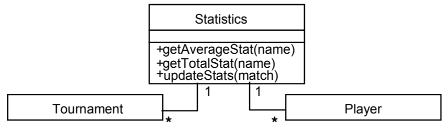
```
public class Player {
    private Map leagues;

    public void addLeague
         (String nickName, League l) {
         if (!leagues.containsKey(l)) {
                   leagues.put(l, nickName);
                   l.addPlayer(nickName, this);
         }
     }
}
```

Transformation of an association class

Object design model before transformation



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



Other Mapping Activities

✓ Optimizing the Object Design Model
✓ Mapping Associations
➢ Mapping Contracts to Exceptions
♦ Mapping Object Models to Tables

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
- Example:
  - **Let us assume the acceptPlayer() operation of TournamentControl is invoked with a player who is already part of the Tournament.**
  - **In this case acceptPlayer() should throw an exception of type KnownPlayer.**
  - **See source code on next slide**

The try-throw-catch Mechanism in Java

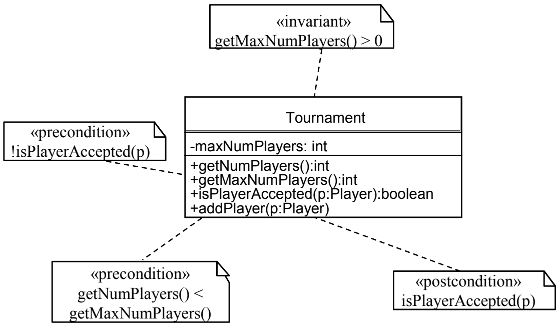
```
public class TournamentControl {
    private Tournament tournament;
    public void addPlayer(Player p) throws KnownPlayerException {
           if (tournament.isPlayerAccepted(p)) {
                      throw new KnownPlayerException(p);
           }
           //... Normal addPlayer behavior
    }
}
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.acceptPlayer((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.

A complete implementation of the Tournament.addPlayer() contract



Heuristics for Mapping Contracts to Exceptions

Be pragmatic, if you don't have enough time.

- Omit checking code for postconditions and invariants.
  - **Usually redundant with the code accomplishing the functionality of the class**
  - **Not likely to detect many bugs unless written by a separate tester.**
- Omit the checking code for private and protected methods.
- Focus on components with the longest life
  - **Focus on Entity objects, not on boundary objects associated with the user interface.**
- Reuse constraint checking code.
  - **Many operations have similar preconditions.**
  - **Encapsulate constraint checking code into methods so that they can share the same exception classes.**

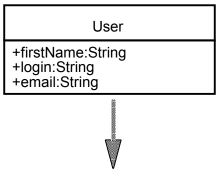
Other Mapping Activities

✓ Optimizing the Object Design Model
✓ Mapping Associations
✓ Mapping Contracts to Exceptions
➢ Mapping Object Models to Tables

Mapping an object model to a relational database

- UML object models can be mapped to relational databases:
  - **Some degradation occurs because all UML constructs must be mapped to a single relational database construct - the table.**
- 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

Mapping the User class to a database table



User table

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

Primary and Foreign Keys

- Any set of attributes that could be used to uniquely identify any data record in a relational table is called a **candidate key**.
- The actual candidate key that is used in the application to identify the records is called the **primary key.**
  - **The primary key of a table is a set of attributes whose values uniquely identify the data records in the table.**
- A **foreign key** is an attribute (or a set of attributes) that references the primary key of another table.

Example for Primary and Foreign Keys

User table

Primary key

firstName	login	email
“alice”	“am384”	“am384@mail.org”
“john”	“js289”	“john@mail.de”
“bob”	“bd”	“bobd@mail.ch”

Candidate key   Candidate key

League table

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

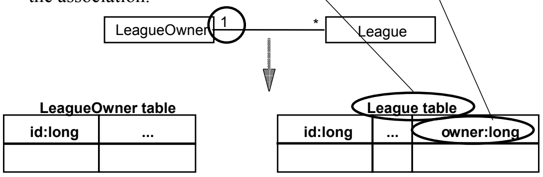
Foreign key referencing **User table**

Buried Association

- Associations with multiplicity one can be implemented using a foreign key.
- For one-to-many associations we add a foreign key to the table representing the class on the “many” end.
- For all other associations we can select either class at the end of the association.

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.
- For all other associations we can select either class at the end of the association.



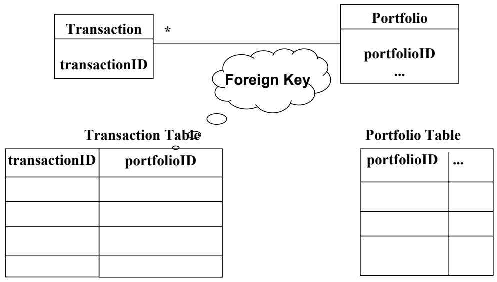
LeagueOwner table

id:long	...

League table

id:long	...	owner:long

Another Example for Buried Association



Transaction Table

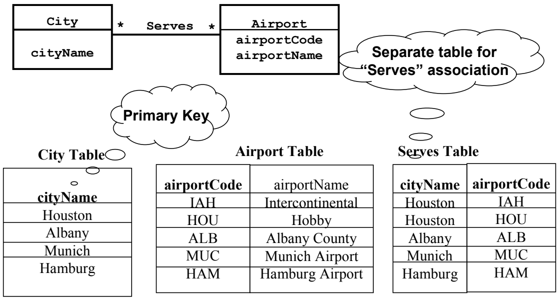
transactionID	portfolioID

Portfolio Table

portfolioID	...

Mapping Many-To-Many Associations

In this case we need a separate table for the association



City Table

cityName
Houston
Albany
Munich
Hamburg

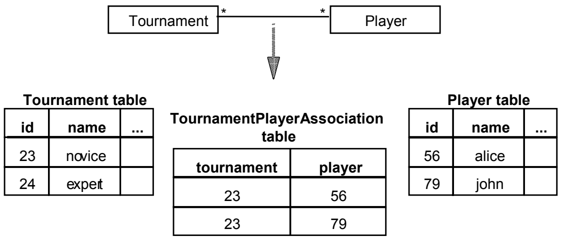
Airport Table

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 the Tournament/Player association as a separate table



Tournament table

id	name	...
23	novice	
24	expet	

TournamentPlayerAssociation table

tournament	player
23	56
23	79

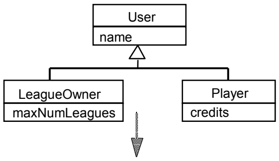
Player table

id	name	...
56	alice	
79	john	

Realizing Inheritance

- Relational databases do not support inheritance
- Two 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**

Realizing inheritance with a separate table



User table

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

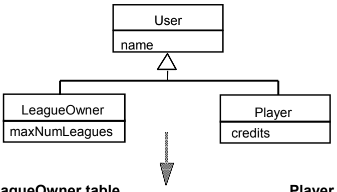
LeagueOwner table

id	maxNumLeagues	...
56	12	

Player table

id	credits	...
79	126	

Realizing 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
  - **How likely is a change of the superclass?**
  - **What are the performance requirements for queries?**
- 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**

Heuristics for Transformations

- For a given transformation use the same tool
  - **If you are using a CASE tool to map associations to code, use the tool to change association multiplicities.**
- Keep the contracts in the source code, not in the object design model
  - **By keeping the specification as a source code comment, they are more likely to be updated when the source code changes.**
- Use the same names for the same objects
  - **If the name is changed in the model, change the name in the code and or in the database schema.**
  - **Provides traceability among the models**
- Have a style guide for transformations
  - **By making transformations explicit in a manual, all developers can apply the transformation in the same way.**

Summary

- Undisciplined changes => degradation of the system model
- Four mapping concepts were introduced
  - **Model transformation improves the compliance of the object design model with a design goal**
  - **Forward engineering improves the consistency of the code with respect to the object design model**
  - **Refactoring improves the readability or modifiability of the code**
  - **Reverse engineering attempts to discover the design from the code.**
- We reviewed model transformation and forward engineering techniques:
  - **Optimizing the class model**
  - **Mapping associations to collections**
  - **Mapping contracts to exceptions**
  - **Mapping class model to storage schemas**