Software Engineering

Professor M. Brian Blake

Lecture 8: Mapping Models to Code

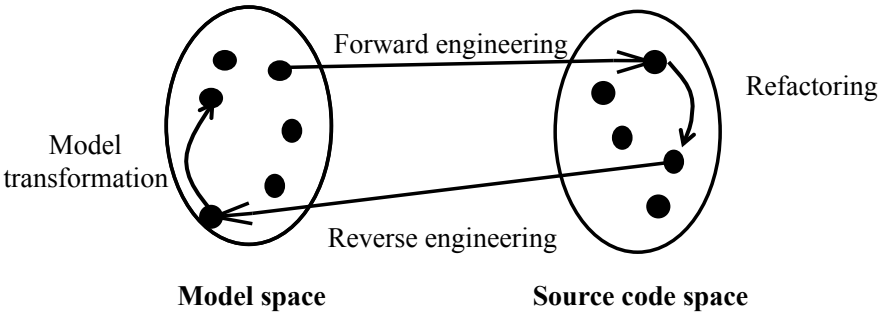
Lecture Objectives

- How do we take what we have learned (OO Modeling techniques) and move it to a design that can be implemented into software?
- How do models fit together?

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
  - 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



Choice of Classes During Analysis Phase

- Domain Classes
  - Classes found in real-world problem domain
  - Carry semantics of the problem
  - Consistent from application to application
  - Very reusable

Choice of Classes During Analysis Phase

- Implementation Classes
  - Implementation-dependent and invisible to the user (stacks, buffers, queues)
  - Avoid these classed in analysis phase
  - But occasionally they are necessary
    - When Specified by Customer
    - Conversion from existing system

Choice of Classes During Analysis Phase

- Application Classes
  - Encapsulate characteristics and behavior of application that are visible to user
  - Represent the intersection of the application and user
  - Built on top of domain model
  - Types
    - Views, Controllers, External Interfaces, Devices, Surrogates, and Metaclasses

Application Classes

- View Classes
  - An external format for presenting information
    - Pictorial, textual, video, audio, report, document
  - Contain both output and input of information
  - Also referred to as *presentation* classes
- Device classes
  - Represent physical devices that support the system (printer, monitor, keyboard, etc.)

Application Classes

- Controller Classes
  - Class that manages an application and its interactions with the outside world
    - User interface controller, scheduler, etc.
  - Multiple controller classes can occur if an application has multiple independent threads of control

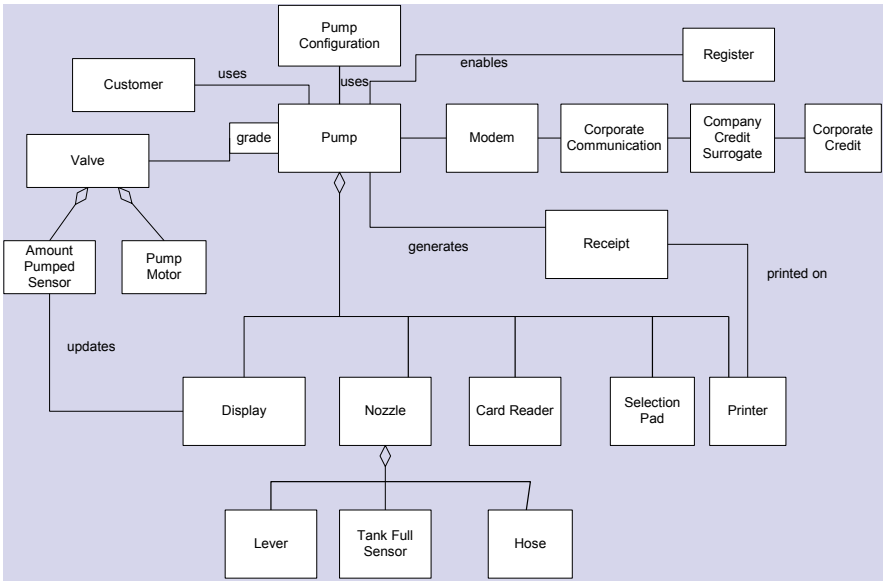
Application Classes

- Interface Classes
  - Class for communicating information to and from an application
  - Mediate all communication with external classes, processes, files and databases
  - Provide a layer of interdependence for the application
  - A mechanism to encapsulate interactions between the subsystems and between applications

Application Classes

- Surrogate Classes
  - Internal representation of real-world external classes
  - Abstracts relevant data from the external class
- Metaclasses
  - Data that defines how a class is defined
  - Useful when a number of classes would otherwise be excessive or rapidly changing

Identifying Class types



Class choices

- View Classes - Display and Receipt
- Controller Class - Pump
- Interface Class - Corporate Communication
- Device Class - Modem
- Surrogate Class - Corp Credit Surrogate
- Metaclass - Pump Configuration

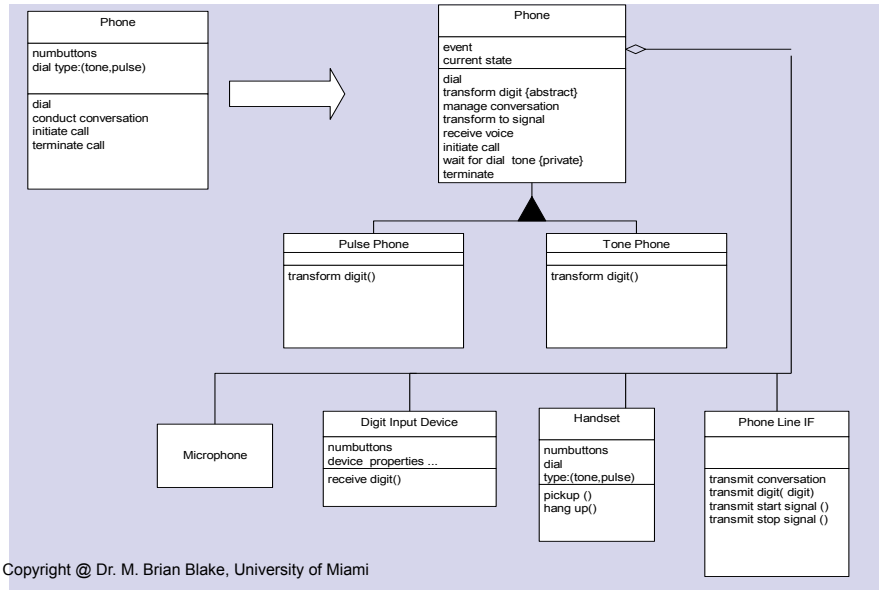
Prior to integrating implementation

- Analysis
  - Create structural (i.e. class diagram), behavioral (i.e. state diagrams), and interaction models (i.e. sequence diagrams)
  - Capture requirements for the domain and application
- System design
  - Defined the subsystems and architectural framework
  - Allocate requirement to individual application programs

Identifying and Mapping Operations

- Identify and Mapping Operations
  - Convert dynamic model features to processes and provide functional descriptions for them
  - Map functional model features into operations and attach them to classes
- This conversion begins the process of mapping the logical structure of the analysis model to the physical structure of the software system to be created
- *Target Classes* - Class chosen to perform actual functionality

Integration Process



Sequence Diagrams

- Functions in Sequence Diagrams translate into real implementation-based functions.

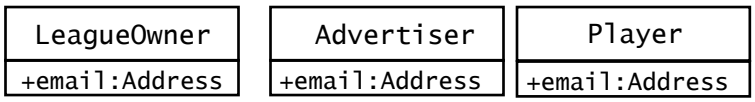
State diagrams

- Convert domain state diagrams to implementation class-based diagrams
  - Events sent by an object represent *operations* on another object
  - Transitions are the change of state on an object
  - State names and Guard conditions represent state attributes on objects

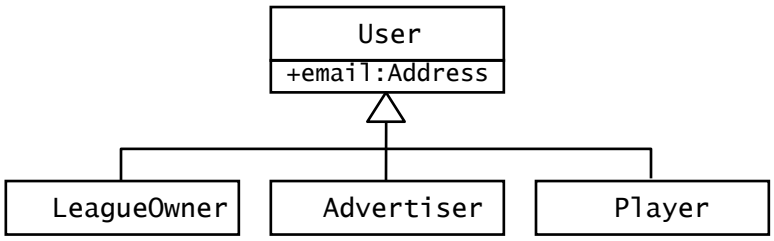
Sample Model Transformations

Model Transformation Example

Object design model before transformation



Object design model after transformation:



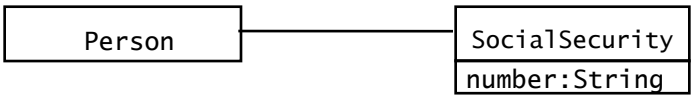
Refactoring Example: Pull Up Field

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

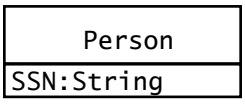
```
public class User {
   private String email;
}
public class Player extends User {
   //...
}
public class LeagueOwner extends User {
   //...
}
public class Advertiser extends User {
   //...
}
```


Collapsing an object without interesting behavior

Object design model before transformation

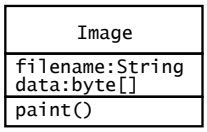


Object design model after transformation

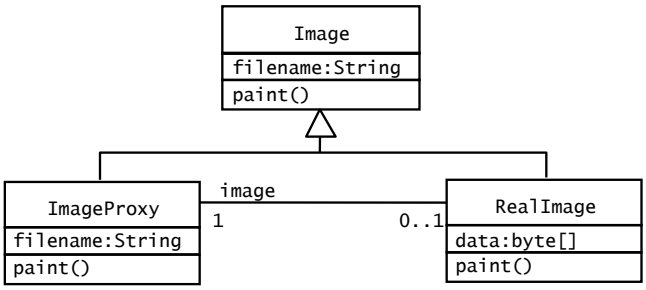


Delaying expensive computations

Object design model before transformation

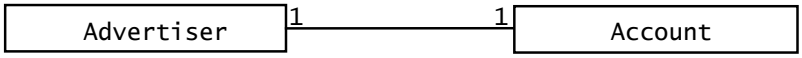


Object design model after transformation



Realization of a unidirectional, one-to-one association

Object design model before transformation

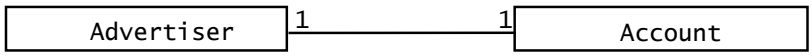


Source code

```
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



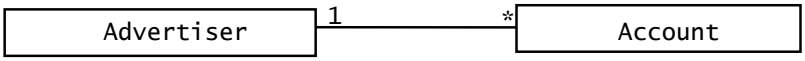
Source code

```
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



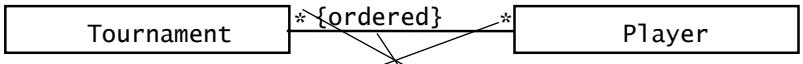
Source code

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

Bidirectional, many-to-many association

Object design model before transformation



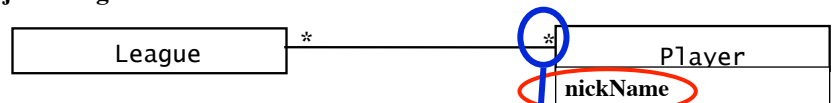
Source code after transformation

```
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);
        }
   }
}
```

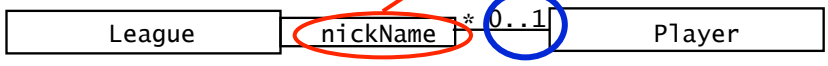
```
public class Player {
   private List tournaments;
   public Player() {
        tournaments = new ArrayList();
   }
   public void addTournament(Tournament t) {
        if (!tournaments.contains(t)) {
         tournaments.add(t);
         t.addPlayer(this);
        }
   }
}
```


Bidirectional qualified association

Object design model before transformation



Object design model before forward engineering



Source code after forward engineering

Bidirectional qualified association (continued)

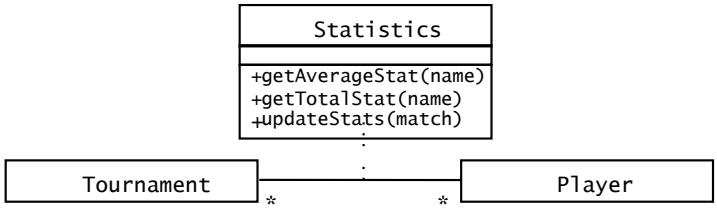
Source code after forward engineering

```
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);
     }
   }
}
```

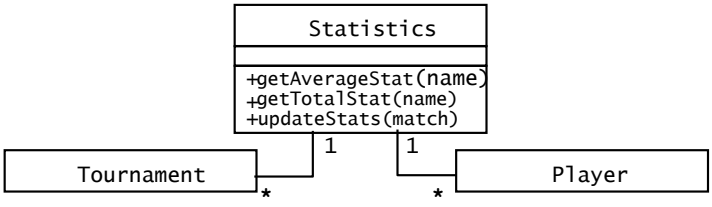
```
public class Player {
   private Map leagues;

   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



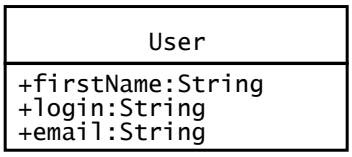
Object design model after transformation: 1 class and two binary associations



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**

Increase Inheritance

- Rearrange and adjust classes and operations to prepare for inheritance
  - Generalization: Finding the base class first, then the sub classes.
  - Specialization: Finding the the sub classes first, then the base class
- Generalization is a common modeling activity. It allows to abstract common behavior out of a group of classes
  - If a set of operations or attributes are repeated in 2 classes the classes might be special instances of a more general class.
- Always check if it is possible to change a subsystem (collection of classes) into a superclass in an inheritance hierarchy.

Create a domain-specific and software-specific class diagram

An Internet Calendar is a personal display of a users daily activities. The Internet calendar displays regular hyperlinks for each activity. Each hyperlink can be pressed to see more detail of the activity. When a user sets up his/her calendar, he/she decides on a calendar year and the calendar software pre-configures the entire year. When the user requires a new entry, there is a pull-down menu to set a specific day/hour, a field for the title, and field for the additional text. The software will create the additional page to hyperlink the additional text. The calendar can only be set by the hour. A user can add/delete as many calendars as he/she likes and can maintain multiple calendars. A calendar can be one of several specialized types, personal business, personal leisure, or group planning.