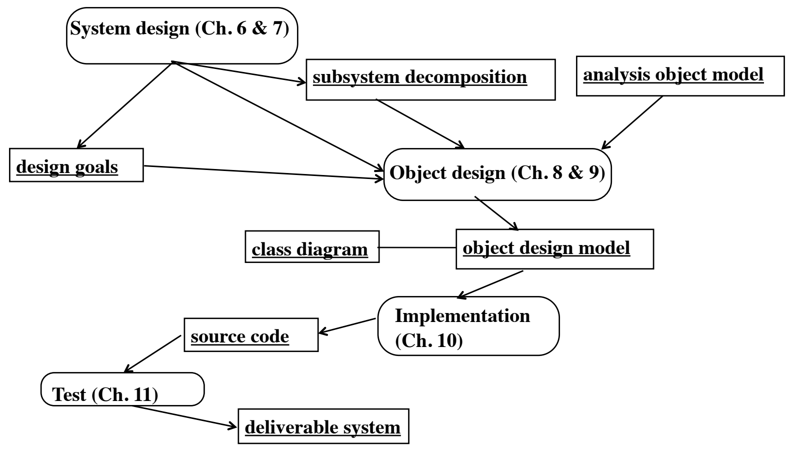
Chapter 10:
Mapping Models to Code

CS 4354
Fall 2012

Jill Seaman

Review: Object-Oriented Development, part 2



10.2 An overview of Mapping

- **Goal**: Describe a selection of transformations to illustrate a disciplined approach to system implementation to avoid system degradation (that results from ad-hoc implementation of the object model)
- **Optimizing the class class model**
  - ✦modifies the object model, to improve performance
- **Mapping Associations to Collections**
- **Mapping Operation Contracts to Exceptions**
  - ✦Implementation of class model components
- **Mapping the class model to a storage schema** (persistence)
  - ✦Storage model for class model components

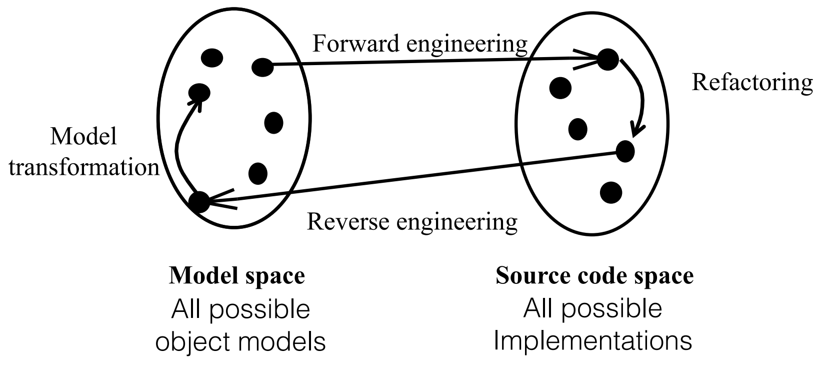
We will focus on these two

10.3 Four types of transformations

A **transformation** aims at improving one aspect of a model (i.e modularity) while preserving all of its other properties (i.e. functionality)

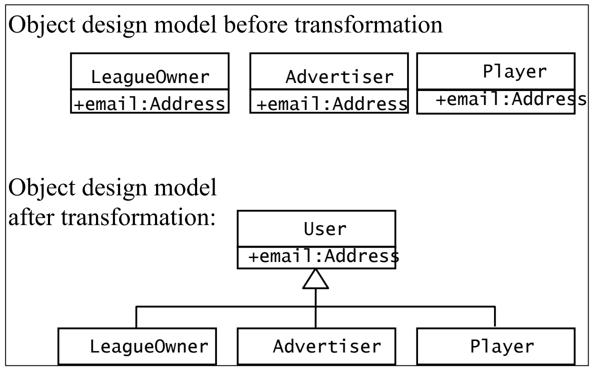
- **Model transformations:** operate on object models.
  - ✦i.e. convert an address attribute to an address class including street address, zip code, city, state, and country attributes.
- **Refactoring:** operate on source code, without changing functionality.
- **Forward engineering:** produces a source code template that corresponds to an object model. (aka implementation)
- **Reverse engineering:** produces a model that corresponds to source code. (often used when design has been lost).

Four types of transformations



10.3.1 Model Transformation

- A **model transformation** is applied to an object model and results in another object model.
  - ✦to simplify or optimize the original model



common attribute is moved to new superclass

10.3.2 Refactoring

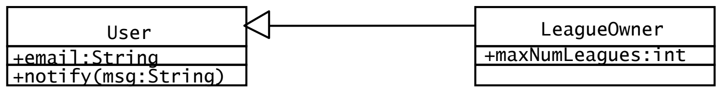
- A **refactoring** is a transformation of the source code that improves its readability or modifiability without changing the behavior of the system.
  - ✦to ensure it does not change behavior: regression tests are run before and after a change
  - ✦changes are kept small in scope
- Example:
  Pull up field is used when two subclasses have the same field.
  - ✦Transformation: move the field(s) to the superclass
  - ✦May need to rename one of the fields first.
  - ✦Compile and test, create new field in superclass, delete from subclasses, compile and test.

10.3.3 Forward Engineering

- **Forward engineering** is applied to a set of model elements and results in a set of corresponding source code statements.
  - ✦to maintain a strong correspondence between the object design model and the code
  - ✦reduce the number of errors introduced during implementation.
- Some examples
  - ✦each UML class maps to a Java class
  - ✦generalization relationship is mapped to “extends superclass” in Java.
  - ✦attributes are mapped to private member variables
    - ➡if the attributes are public, then public getter and setter methods are defined as members of the class.

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 */
}
```

10.3.4 Reverse Engineering

- **Reverse engineering** is applied to a set of source code statements and results in a set of corresponding model elements.
  - ✦to recreate the model for an existing system, because it was lost or never created
  - ✦essentially the inverse of forward engineering
- Some examples
  - ✦create a UML class for each Java class
  - ✦create an attribute for each field in the class
  - ✦create an operation for each member function in the class
    - ➡Will not necessarily create the same model as the original (if the code was derived from a UML model).

10.3.5 Transformation Principles

- In performing transformations, we are trying to improve one aspect of the system, but we run the risk of introducing errors that are difficult to detect and repair. So we follow these principles:
- Each transformation must address a **single** criteria (design goal)
  - ✦For example, to improve response time or to improve coherence.
- Each transformation must be local.
  - ✦Change only a few methods or classes at once.
- Each transformation must be applied in isolation to other changes.
  - ✦i.e. don't try to add functionality when refactoring,
- Each transformation must be followed by a validation step.
  - ✦run regression tests, update sequence diagrams, check use cases

10.4 Mapping Activities
10.4.1 Optimizing the Object Design Model

- Direct translation of analysis model to source code often results in inefficient code
  - ✦analysis model focuses on the functionality of the system
  - ✦it does not take into account system design decisions.
- Object design: transform the object model to meet the design goals: minimize response time, execution time, or memory resources.
- Four common optimizations:
  - ✦adding associations to optimize access paths
  - ✦collapsing objects into attributes
  - ✦delaying expensive computations (proxy design pattern)
  - ✦caching the results of expensive computations.

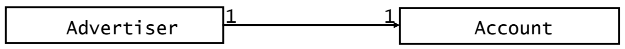
See book for details

10.4.2 Mapping Associations to Collections

- Object-oriented languages do not support “associations” (which are usually bidirectional) between objects.
  - ✦Instead one class can contain a single reference or a collection of references to (an)other object(s). These are unidirectional
- We will demonstrate mapping the following associations to Java:
  - ✦Unidirectional one-to-one associations
  - ✦Bidirectional one-to-one associations.
  - ✦Bidirectional one-to-many associations
  - ✦Bidirectional many-to-many associations

Unidirectional one-to-one association

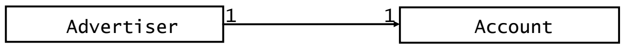
- Object design model before transformation (note arrow on assoc.)



- Add (private) field to Advertiser class to refer to an Account object
- Add getAccount() and setAccount(Account a) to Advertiser
- IF the Account field never changes:
  - ✦ don’t have a setter (setAccount)
  - ✦ create the instance of Account in Advertiser’s constructor

Unidirectional one-to-one association example

- Object design model before transformation (note arrow on assoc.)



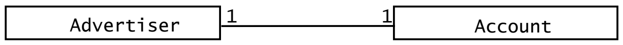
- Source code of Advertiser after transformation (assuming the Account object never changes)

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

Create Account in constructor
No setAccount()

Bidirectional one-to-one association

- Object design model before transformation (no arrow)



- Add (private) field to Advertiser class to refer to an Account object
  Add (private) field to Account class to refer to an Advertiser object
- Add getters and setters to both classes
  BUT the setters must maintain this constraint (bidirectionality):
  a.getAccount() equals b <==> b.getAdvertiser() equals a

```
public void setAccount(Account b) {
  if (account != b) {
    Account old = account;
    account = b;
    if (old!=null)
      old.setAdvertiser(null);
    if (b!=null)
      b.setAdvertiser(this);
}  }
```

```
public void setAdvertiser(Advertiser a){
  if (advertiser != a) {
    Advertiser old = advertiser;
    advertiser = a;
    if (old!=null)
      old.setAccount(null);
    if (a!=null)
      a.setAccount(this);
} }
```

Bidirectional one-to-one association example

- IF the Account and Advertiser fields never change:
  - ✦ don't have setter methods for these
  - ✦ create the instance of Account in Advertiser's constructor (pass **this** to it)
  - ✦ set advertiser in the Account constructor.

```
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;
  }
}
```

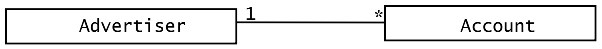
```
public class Account {
  /* The advertiser field is
   * initialized in the
   * constructor and
   * never modified. */
  private Advertiser advertiser;

  public Account(Advertiser a) {
      this.advertiser = a;
  }
  public Advertiser getAdvertiser() {
      return advertiser;
  }
}
```

Note: classes are mutually dependent

Bidirectional one-to-many association

- Object design model before transformation (note asterisk)



- Add field to Advertiser to refer to a collection of Account objects
- Add field to Account class to refer to an Advertiser object
- Add: addAccount(a) and removeAccount(a) to Advertiser
- Add: getAdvertiser and setAdvertiser(b) to Account
- BUT the mutators must maintain this constraint (bidirectionality):

a.getAccounts() contains b <==> b.getAdvertiser() equals a

Bidirectional one-to-many association example

- Source code after transformation

```
public class Advertiser {
  private Set accounts;
                  //A Set of references to Account
  public Advertiser() {
      accounts = new HashSet();
  }
   public Set getAccounts() {
      return accounts;
  }
  public void addAccount(Account a) {
      accounts.add(a);
      a.setAdvertiser(this);
  }
  public void removeAccount(Account a) {
      accounts.remove(a);
      a.setAdvertiser(null);
  }
}
```

```
public class Account {
  private Advertiser advertiser;
  public Advertiser
     getAdvertiser() {
       return advertiser;
  }
  public void setAdvertiser
   (Advertiser n) {
     if (advertiser != n) {
       Advertiser old = advertiser;
       advertiser = n;
       if (old != null)
          old.removeAccount(this);
       if (n != null)
          n.addAccount(this);
     }
  }
}
```

Bidirectional one-to-many association example

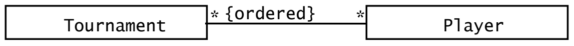
- IF the Account and Advertiser fields never change:
  - ✦ don't have mutator methods for these fields
  - ✦ set advertiser from the Account constructor (has Advertiser parameter)
  - ✦ objects of both types are created and connected by a controller class

```
public class Advertiser {
  private Set accounts;
  public Advertiser() {
      accounts = new HashSet();
  }
   public Set getAccounts() {
      return accounts;
  }
   //call only from Account constructor
  public void addAccount(Account a) {
      accounts.add(a);
  }
}
```

```
public class Account {
  private Advertiser advertiser;
  //should call constructor ONLY
  //from Advertiser class
  public Account (Advertiser n) {
    advertiser = n;
    n.addAccount(this);
  }
  public Advertiser getAdvertiser()
  {
      return advertiser;
  }
}
```

Bidirectional many-to-many association

- Object design model before transformation (note two asterisks)



- Add field to Tournament to refer to a collection of Player objects
- Add field to Player to refer to a collection of Tournament objects
- Add: addPlayer(p) and removePlayer(a) to Tournament
- Add: addTournament(t) and removeTournament(t) to Player
- BUT the mutators must maintain this constraint (bidirectionality):

t.getPlayers() contains p <==> p.getTournaments() contains t

Bidirectional many-to-many association example

```
public class Tournament {
  private List players;
  public Tournament() {
      players =
         new ArrayList<Player>();
  }
  public void addPlayer(Player p)
    {
      if (!players.contains(p)){
         players.add(p);
         p.addTournament(this);
      }
  }
   public void remPlayer(Player p)
    {
      if (players.contains(p)){
         players.remove(p);
         p.remTournament(this);
      }
  }
}
```

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

10.4.3 Mapping Contracts to Exceptions

- 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
  - ✦For example, let's assume we want to enforce the following precondition for the addPlayer() operation of class Tournament

```
context Tournament::addPlayer(p) pre:
 not isPlayerAccepted(p)
```

  - ✦We can add the following code to the beginning of the addPlayer method:

```
public void addPlayer(Player p) throws KnownPlayerException {
  if (isPlayerAccepted(p)) {
      throw new KnownPlayerException(p);
  }
  //... Normal addPlayer behavior
}
```

Simple Mapping of Contracts to Exceptions

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 with the postconditions.
- **Deal with inheritance:** Encapsulate the checking code for preconditions and postconditions into separate methods that can be called from subclasses.

Example: Tournament::addPlayer

```
public class Tournament {
//…
  private List players;
  public void addPlayer (Player P) throws KnownPlayer, TooMany Players,
  UnknownPlayer, IllegalNumPlayers, IllegalMaxNumPlayers  {

  // check precondition : !isPlayerAccepted (p)
   if (isPlayerAccepted(p) {
        throw new KnownPlayer(p);
    }
  // check precondition: getNumPlayers() < maxNumPlayers
    if (getNumPlayers() == getMaxNumPlayers()) {
         throw new TooManyPlayers (getNumPlayers());
    }
   // save values for postconditions
     int pre_getNumPlayers = getNumPlayers ();
   // accomplish the real work
     players.add(p);
     p.addTournament(this);
   // check post condition isPlayerAccepted(p)
      if (!isPlayerAccepted(p)) {
          throw new UnknownPlayer(p);
    }
   // check post condition getNumPlayers() = pre.getNumPlayers( ) + 1
      if (getNumPlayers( ) != pre_getNumPlayers+1) {
         throw new IllegalNumPlayers (getNumPlayers( ));
     }
    // check invariant maxNumPlayers > 0
      if (getMaxNumPlayers ( ) <= 0) {
         throw IllegalMaxNumPlayers (getMaxNumPlayers ( ));
      }
  }
// …
}
```

Mapping of Contracts to Exceptions IRL

Systematic application of rules leads to a robust system, but it is NOT realistic:

- **Coding Effort:** Checking code is longer and more complex than code accomplishing the real work
- **Increased opportunities for defects:** Checking code itself may have errors, and may mirror errors in code doing real work.
- **Obfuscated code:** Checking code is more complex and difficult to modify when constraints change.
- **Performances drawback:** Checking all contracts systematically can significantly slow down the code.

Use heuristics to decide when implementing contracts is worthwhile

10.4.4 Mapping Object Models to a Persistent Storage Schema

- UML object models can be mapped to relational databases:
  - ✦A relational database is basically a set of tables
  - ✦A foreign key is a column used to reference a row in another 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 one-to-many association is implemented using a foreign key
  - ✦A many-to-many association is mapped into its own table, using foreign keys
- Methods are not mapped

See book for details