Mapping Models to Code

• We will skip most of this chapter
  – It deals with how to go from UML class diagrams to actual code
  – You should already have a pretty good idea how to do this
    • Would be trickier if using a language that doesn’t support OOP for example

• Overview
  – Mappings are transformations that aim at improving one aspect of the model while preserving functionality.
  Activities:
    • Optimization
    • Realizing associations
    • Contracts to exceptions
    • Class models to storage schema
Transformations

Model transformation

Model space

Source code space

Forward engineering

Reverse engineering

Refactoring

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
```
Refactoring Example: Pull Up Field

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

```java
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 extends User {
  public Advertiser(String email) {
    this.email = email;
  }
}
```

---

Refactoring Example: Pull Up Constructor Body

```java
public class User {
  private String 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

```
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 */
}
```

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 */
}
```

Transformation Principles

- Each transformation must address a single criteria
  - Transformation should focus on a single design goal and not try to optimize multiple criteria (can lead to errors)

- Each transformation must be local
  - A transformation should change only a few methods or classes at once
  - If an interface changes then the client classes should be changed now too (keep older method around for background compatibility testing)
  - If you are changing many subsystems at once you are performing an architectural change
Transformation Principles

• Each transformation must be applied in isolation to other changes
  – To localize changes transformations should be applied one at a time
  – E.g. if improving performance of a method, don’t add new functionality at the same time

• Each transformation must be followed by a validation step
  – Validate the changes for errors
  – Update appropriate UML diagrams
  – Write new test cases to exercise new source code