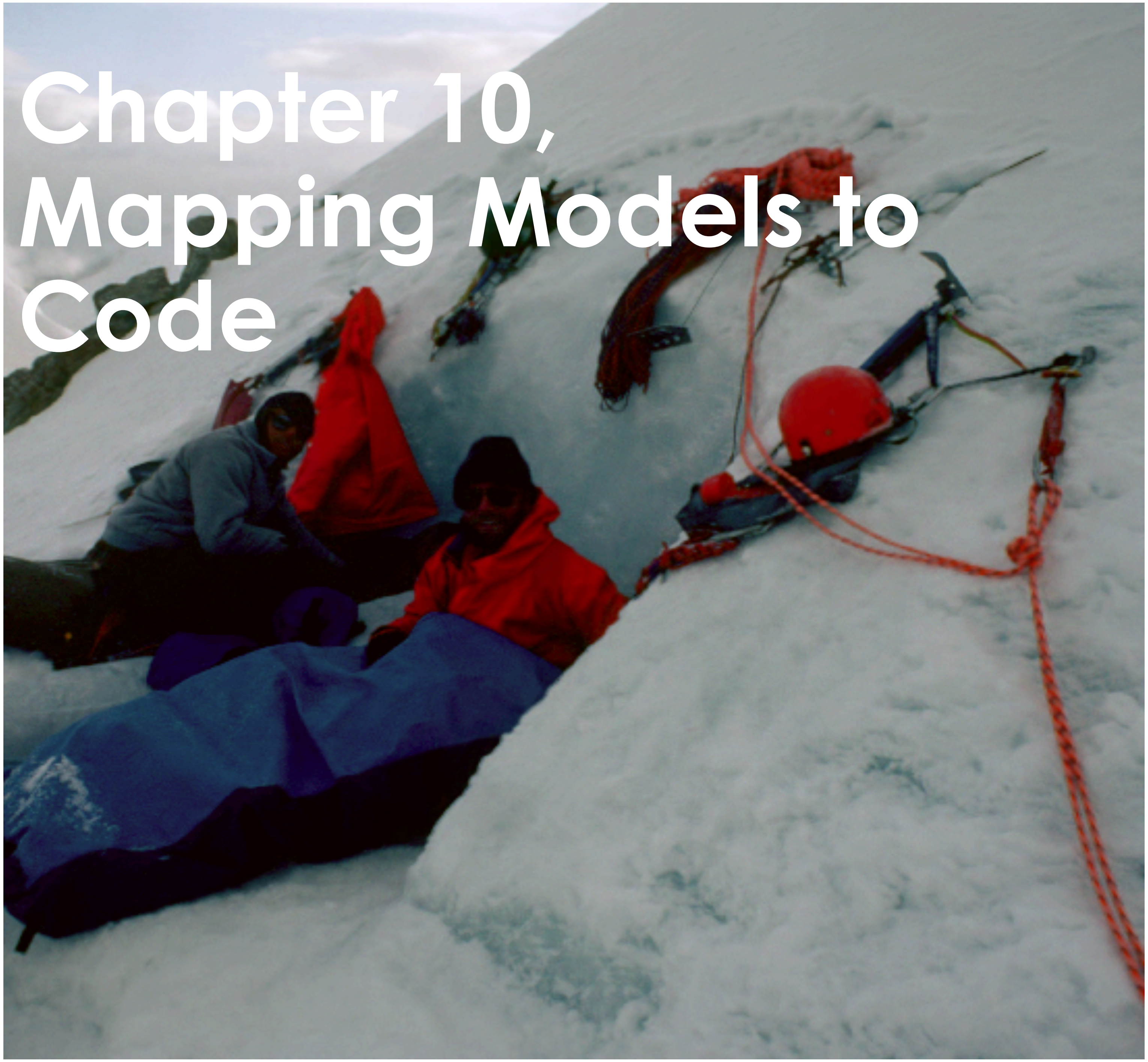
Object-Oriented Software Engineering
Using UML, Patterns, and Java



Chapter 10, Mapping Models to Code

Lecture Plan

- Part 1
  - Implementation of class model components:
    - Realization of associations
    - Realization of operation contracts
- Part 2
  - Realizing entity objects based on selected storage strategy
  - Mapping the object model to a storage schema
  - Mapping class diagrams to tables

Problems with implementing an Object Design Model

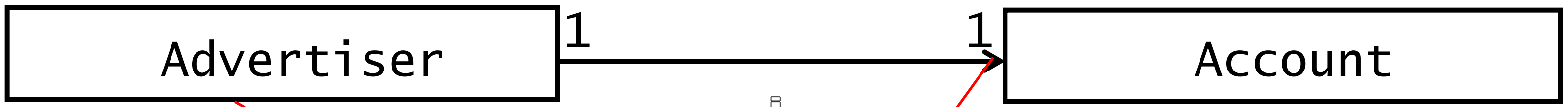
- Programming languages do not support the concept of UML associations
  - The associations of the object model must be transformed into collections of object references
- Many programming languages do not support contracts (invariants, pre and post conditions)
  - Developers must therefore manually transform contract specification into source code for detecting and handling contract violations
- The client changes the requirements during object design
  - The developer must change the contracts in which the classes are involved
- All these object design activities cause problems, because they need to be done manually.

Mapping Associations

1. Unidirectional one-to-one association
2. Bidirectional one-to-one association
3. Bidirectional one-to-many association
4. Bidirectional many-to-many association
5. Bidirectional qualified association.

Unidirectional one-to-one association

Object design model before transformation:

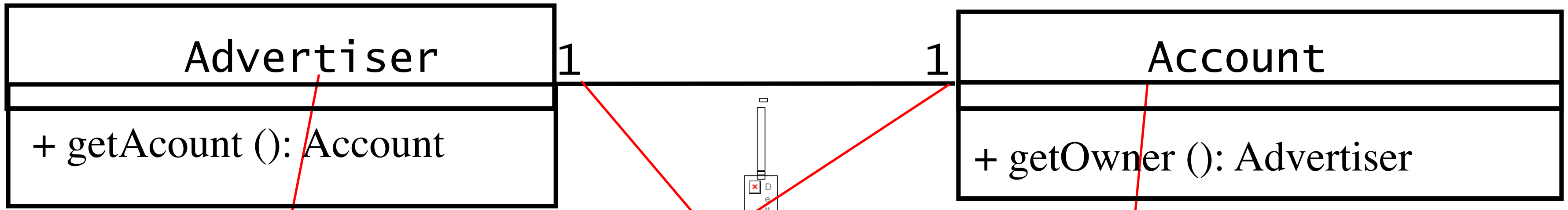


Source code after transformation:

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

Bidirectional one-to-one association

Object design model before transformation:



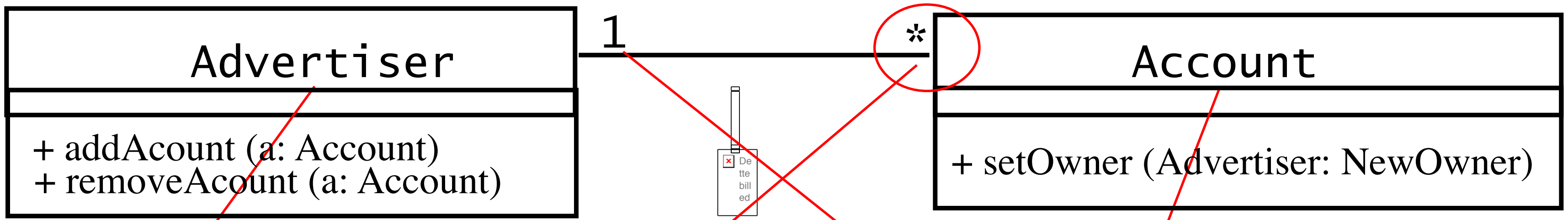
Source code after transformation:

```
public class Advertiser {
/* account 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 {
  /* owner is initialized
   * in 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:


Advertiser
+ addAcount (a: Account)
+ removeAcount (a: Account)
1
*
Account
+ setOwner (Advertiser: NewOwner)

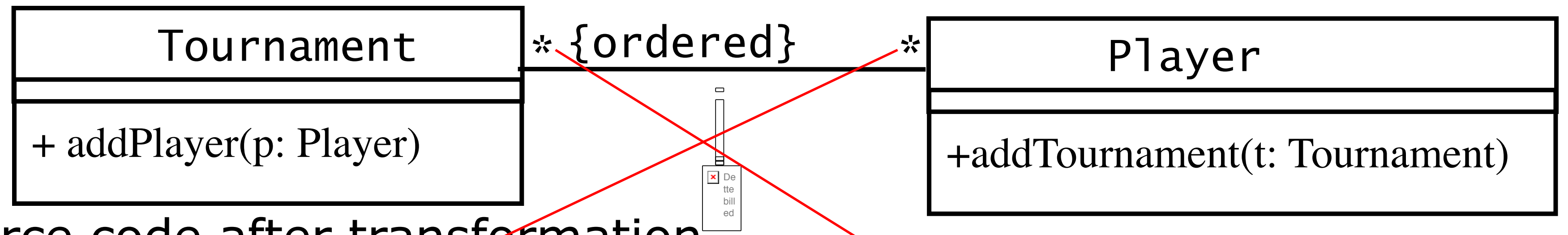
Source code after transformation:

```
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);
     }
   }
}
```

Examples of Model Transformations and Forward Engineering

- Model Transformations
  - Goal: Optimizing the object design model
    - ✓ Collapsing objects
    - ✓ Delaying expensive computations
- Forward Engineering
  - Goal: Implementing the object design model in a programming language
  - ✓ Mapping inheritance
  - ✓ Mapping associations
  - Mapping contracts to exceptions ➡ Moved to Exercise Session on Testing
  - Next! ➡ Mapping object models to tables

Summary

- Strategy for implementing associations:
  - Be as uniform as possible
  - Individual decision for each association
- Example of uniform implementation
  - 1-to-1 association:
    - Role names are treated like attributes in the classes and translate to references
  - 1-to-many association:
    - "Ordered many" : Translate to `Vector`
    - "Unordered many" : Translate to `Set`
  - Qualified association:
    - Translate to Hash table

Heuristics for Implementing Associations

- Two strategies for implementing associations:
  1. Be as uniform as possible
  2. Make an individual decision for each association
- Example of a uniform implementation (often used by CASE tools)
  - 1-to-1 association:
    - Role names are always treated like attributes in the classes and translated into references
  - 1-to-many association:
    - Always translated into a Vector
  - Qualified association:
    - Always translated into to a Hash table.

Model-Driven Engineering

- http://en.wikipedia.org/wiki/Model_Driven_Engineering
- Model-driven engineering refers to a range of development approaches that are based on the use of software modeling as a primary form of expression. Sometimes models are constructed to a certain level of detail, and then code is written by hand in a separate step.
- Sometimes complete models are built including executable actions. Code can be generated from the models, ranging from system skeletons to complete, deployable products.
- With the introduction of the Unified Modeling Language (UML), MDE has become very popular today with a wide body of practitioners and supporting tools. More advanced types of MDE have expanded to permit industry standards which allow for consistent application and results. The continued evolution of MDE has added an increased focus on architecture and automation.
- MDE technologies with a greater focus on architecture and corresponding automation yield higher levels of abstraction in software development. This abstraction promotes simpler models with a greater focus on problem space. Combined with executable semantics this elevates the total level of automation possible.
- The Object Management Group (OMG) has developed a set of standards called model-driven architecture (MDA), building a foundation for this advanced architecture-focused approach.