Mapping Models to Java Code

Introduction into Software Engineering
Lecture 16

Bernd Bruegge
Applied Software Engineering
Technische Universitaet Muenchen
Lecture Plan

• Part 1
  • Operations on the object model:
    • Optimizations to address performance requirements
  • 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
Characteristics of Object Design Activities

- Developers try to improve modularity and performance
- Developers need to transform associations into references, because programming languages do not support associations
- If the programming language does not support contracts, the developer needs to write code for detecting and handling contract violations
- Developers need to revise the interface specification whenever the client comes up with new requirements.
State of the Art: Model-based Software Engineering

• The Vision
  • During object design we build an object design model that realizes the use case model and which is the basis for implementation (model-driven design)

• The Reality
  • Working on the object design model involves many activities that are error prone
  • Examples:
    • A new parameters must be added to an operation. Because of time pressure it is added to the source code, but not to the object model
    • Additional attributes are added to an entity object, but not handled by the data management system (thus they are not persistent).
Other Object Design Activities

• Programming languages do not support the concept of a UML association
  • 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 interface specification of the involved classes

• All these object design activities cause problems, because they need to be done manually.
• Let us get a handle on these problems
• To do this we distinguish two kinds of spaces
  • the model space and the source code space
• and 4 different types of transformations
  • Model transformation
  • Forward engineering
  • Reverse engineering
  • Refactoring.
4 Different Types of Transformations

- **Model space**
  - Model transformation
  - System Model (in UML)
  - Another System Model
  - Yet Another System Model

- **Source code space**
  - Forward engineering
  - Refactoring
  - Reverse engineering
  - Program (in Java)
  - Another Program

© 2007 Bernd Bruegge

Introduction into Software Engineering  Summer 2007
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
4 Different Types of **Transformations**

**Source code space**
- Forward engineering
- Refactoring

**Model space**
- Reverse engineering
- Model transformation

- System Model (in UML)
- Another System Model
- Yet Another System Model

- Program (in Java)
- Another Program

© 2007 Bernd Bruegge
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;
    //...
}

public class User {
    private String email;
}

public class Player extends User {
    //...
}

public class LeagueOwner extends User {
    User {
        //...
    }

public class Advertiser extends User {
    User {
        //...
    }
```
Refactoring Example: Pull Up Constructor Body

```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;
    }
}
```

```java
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);
    }
}
```
4 Different Types of Transformations

Model space

Source code space

Model transformation:
- System Model (in UML)

Forward engineering:
- Yet Another System Model

Reverse engineering:
- Other Program

Refactoring:
- Program (in Java)

Another System Model
Forward Engineering Example

Object design model before transformation:

<table>
<thead>
<tr>
<th>User</th>
<th>LeagueOwner</th>
</tr>
</thead>
<tbody>
<tr>
<td>-email: String</td>
<td>-maxNumLeagues:int</td>
</tr>
<tr>
<td>+getEmail():String</td>
<td>+getMaxNumLeagues():int</td>
</tr>
<tr>
<td>+setEmail(e: String)</td>
<td>+setMaxNumLeagues(n: int)</td>
</tr>
<tr>
<td>+notify(msg: String)</td>
<td></td>
</tr>
</tbody>
</table>

Source code after transformation:

```java
public class User {
    private String email;
    public String getEmail() {
        return email;
    }
    public void setEmail(String value) {
        email = value;
    }
    public void notify(String msg) {
        // ....
    }
}

public class LeagueOwner extends User {
    private int maxNumLeagues;
    public int getMaxNumLeagues() {
        return maxNumLeagues;
    }
    public void setMaxNumLeagues(int value) {
        maxNumLeagues = value;
    }
}
```
More 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
    - Mapping object models to tables
Collapsing Objects

Object design model before transformation:

```
Person

SocialSecurity
number:String
```

Object design model after transformation:

```
Person

SSN:String
```

Turning an object into an attribute of another object is usually done, if the object does not have any interesting dynamic behavior (only get and set operations).
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
    • Mapping object models to tables
Delaying expensive computations

Object design model before transformation:

Object design model after transformation:

Proxy Pattern!
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
    • Mapping object models to tables
Forward Engineering: Mapping a UML Model into Source Code

• **Goal:** We have a UML-Model with inheritance. We want to translate it into source code

• **Question:** Which mechanisms in the programming language can be used?
  • Let’s focus on Java

• Java provides the following mechanisms:
  • Overwriting of methods (default in Java)
  • Final classes
  • Final methods
  • Abstract methods
  • Abstract classes
  • Interfaces.
Realizing Inheritance in Java

- Realisation of specialization and generalization
  - Definition of subclasses
  - Java keyword: `extends`

- Realisation of simple inheritance
  - Overwriting of methods is not allowed
  - Java keyword: `final`

- Realisation of implementation inheritance
  - Overwriting of methods
  - No keyword necessary:
    - Overwriting of methods is default in Java

- Realisation of specification inheritance
  - Specification of an interface
  - Java keywords: `abstract`, `interface

See Slide 13
Example for the use of Abstract Methods: Cryptography

• Problem: Delivery a general encryption method
• Requirements:
  • The system provides algorithms for existing encryption methods (e.g. Caesar, Transposition)
  • New encryption algorithms, when they become available, can be linked into the program at runtime, without any need to recompile the program
  • The choice of the best encryption method can also be done at runtime.
Object Design of Chiffre

- We define a super class Chiffre and define subclasses for the existing existing encryption methods

- 4 public methods:
  - `encrypt()` encrypts a text of words
  - `decrypt()` deciphers a text of words
  - `encode()` uses a special algorithm for encryption of a single word
  - `decode()` uses a special algorithm for decryption of a single word.
Implementation of Chiffre in Java

• The methods `encrypt()` and `decrypt()` are the same for each subclass and can therefore be implemented in the superclass `Chiffre`  
  • `Chiffre` is defined as subclass of `Object`, because we will use some methods of `Object`

• The methods `encode()` and `decode()` are specific for each subclass  
  • We therefore define them as `abstract methods` in the super class and expect that they are implemented in the respective subclasses.

Exercise: Write the corresponding Java Code!
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
    • Mapping object models to tables
Mapping Associations

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

Object design model before transformation:

```plaintext
Account
   └── Advertiser
       └── 1
```

Source code after transformation:

```java
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:

```
Account 1
   --------------
Advertiser 1
   --------------
Account
```

Source code after transformation:

```java
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;
   }
}
```

```java
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:

![Diagram of object design model]

Source code after transformation:

```java
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);
    }
}
```

```java
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

```
Tournament * {ordered} * Player
```

Source code after transformation

```java
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);
        }
    }
}
```

```java
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 model transformation

```
League [*] Player [*]
  nickName
```

Object design model after model transformation

```
League [*] nickName [0..1] Player [*]
```

Source code after forward engineering (see next slide 31)
Bidirectional qualified association (2)

Object design model before forward engineering

```
League

nickName

Player
```

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);
        }
    }
}
```
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
  • Mapping object models to tables
Implementing Contract Violations

• Many object-oriented languages do not have built-in support for contracts
• However, if they support exceptions, we can use their exception mechanisms 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
    • UML model (see slide 34)
  • In this case acceptPlayer() in TournamentControl should throw an exception of type KnownPlayer
    • Java Source code (see slide 35).
UML Model for Contract Violation Example

TournamentForm

+applyForTournament()

TournamentControl

+selectSponsors(advertisers):List
+advertizeTournament()
+acceptPlayer(p)
+announceTournament()
+isPlayerOverbooked():boolean

Tournament

- maNumPlayers:String
+ start:Date
+ end:Date
+ acceptPlayer(p)
+ removePlayer(p)
+ isPlayerAccepted(p)

Player

* players

* matches

Match

+ start:Date
+ status:MatchStatus
+ playMove(p,m)
+ getScore():Map

* matches

TournamentForm

TournamentControl

* * players

* Advertiser

* * sponsors
public class TournamentForm {
    private TournamentControl control;
    private ArrayList players;
    public void processPlayerApplications() {
        for (Iteration i = players.iterator(); i.hasNext();)
            try {
                control.acceptPlayer(((Player)i.next()));
            } catch (KnownPlayerException e) {
                // If exception was caught, log it to console
                ErrorConsole.log(e.getMessage());
            }
    }
}
The try-throw-catch Mechanism in Java

```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() {
        for (Iteration i = players.iterator(); i.hasNext(); ) {
            try {
                control.acceptPlayer((Player)i.next());
            } catch (KnownPlayerException e) {
                // If exception was caught, log it to console
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}
```
TournamentControl

- applyForTournament()

TournamentForm

- applyForTournament()

Tournament

- maNumPlayers: String
  - start: Date
  - end: Date

Player

- matches

Match

- matches

Adviser

- sponsors

Sponsors

- selectSponsors(advertisers): List
  - advertiseTournament()
  - acceptPlayer(p)
  - announceTournament()
  - isPlayerOverbooked(): Boolean

Player

- matches

Tournament

- maNumPlayers: String
  - start: Date
  - end: Date

Player

- matches

Match

- matches

Adviser

- sponsors

Sponsors

- selectSponsors(advertisers): List
  - advertiseTournament()
  - acceptPlayer(p)
  - announceTournament()
  - isPlayerOverbooked(): Boolean
Implementing a Contract

- **Check each precondition:**
  - Before the beginning of the method with a test to check the precondition for that method
    - Raise an exception if the precondition evaluates to false

- **Check each postcondition:**
  - At the end of the method write a test to check the postcondition
    - Raise an exception if the postcondition evaluates to false. If more than one postcondition is not satisfied, raise an exception only for the first violation.

- **Check each invariant:**
  - Check invariants at the same time when checking preconditions and when checking postconditions

- **Deal with inheritance:**
  - Add the checking code for preconditions and postconditions also into methods that can be called from the class.
A complete implementation of the Tournament.addPlayer() contract

```
<invariant>
getMaxNumPlayers() > 0
```

```
Tournament

-maxNumPlayers: int
+getNumPlayers():int
+getMaxNumPlayers():int
+isPlayerAccepted(p:Player):boolean
+addPlayer(p:Player)
```

```
<precondition>
!isPlayerAccepted(p)
```

```
<precondition>
getNumPlayers() < getMaxNumPlayers()
```

```
<postcondition>
isPlayerAccepted(p)
```
Heuristics: Mapping Contracts to Exceptions

• Executing checking code slows down your program
  • If it is too slow, omit the checking code for private and protected methods
  • If it is still too slow, focus on components with the longest life
    • Omit checking code for postconditions and invariants for all other components.
Heuristics for Transformations

• For any given transformation always use the same tool
• Keep the contracts in the source code, not in the object design model
• Use the same names for the same objects
• Have a style guide for transformations (Martin Fowler)
Summary

• Four mapping concepts:
  • Model transformation
  • Forward engineering
  • Refactoring
  • Reverse engineering

• Model transformation and forward engineering techniques:
  • Optizimizing the class model
  • Mapping associations to collections
  • Mapping contracts to exceptions
  • Mapping class model to storage schemas
Backup and Additional Slides
Transformation of an Association Class

Object design model before transformation

Object design model after transformation: 1 class and 2 binary associations
More Terminology

• **Roundtrip Engineering**
  - Forward Engineering + reverse engineering
  - Inventory analysis: Determine the Delta between Object Model and Code
  - Together-J and Rationale provide tools for reverse engineering

• **Reengineering**
  - Used in the context of project management:
  - Providing new functionality (customer dreams up new stuff) in the context of new technology (technology enablers)
Specifying Interfaces

• The players in object design:
  • Class User
  • Class Implementor
  • Class Extender

• Object design: Activities
  • Adding visibility information
  • Adding type signature information
  • Adding contracts

• Detailed view on Design patterns
  • Combination of delegation and inheritance
Statistics as a product in the Game Abstract Factory

Tournament → Game

Game

createStatistics()

TicTacToeGame

ChessGame

Statistics

update()

getStat()

TTTStatistics

ChessStatistics

DefaultStatistics
N-ary association class Statistics

Statistics relates League, Tournament, and Player
Realization of the Statistics Association

```
update(match)
getStatNames(game)
getStat(name, game, player)
getStat(name, league, player)
getStat(name, tournament, player)
```

```
update(match, player)
gStatNames()
gStat(name)
```

```
createStatistics()
```
StatisticsVault as a Facade

TournamentControl

StatisticsView

StatisticsVault

update(match)
getStatNames(game)
getStat(name,game,player)
getStat(name,league,player)
getStat(name,tournament,player)

class Statistics

update(match,player)
getStatNames()
getStat(name)

class Game

class createStatistics()
Public interface of the StatisticsVault class

```java
public class StatisticsVault {
    public void update(Match m)
        throws InvalidMatch, MatchNotCompleted {...}

    public List getStatNames() {...}

    public double getStat(String name, Game g, Player p)
        throws UnknownStatistic, InvalidScope {...}

    public double getStat(String name, League l, Player p)
        throws UnknownStatistic, InvalidScope {...}

    public double getStat(String name, Tournament t, Player p)
        throws UnknownStatistic, InvalidScope {...}
}
```
Database schema for the Statistics Association

<table>
<thead>
<tr>
<th>Statistics table</th>
</tr>
</thead>
<tbody>
<tr>
<td>id:long</td>
</tr>
<tr>
<td>scope:long</td>
</tr>
<tr>
<td>scopetype:long</td>
</tr>
<tr>
<td>player:long</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StatisticCounters table</th>
</tr>
</thead>
<tbody>
<tr>
<td>id:long</td>
</tr>
<tr>
<td>name:text[25]</td>
</tr>
<tr>
<td>value:double</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Game table</th>
</tr>
</thead>
<tbody>
<tr>
<td>id:long</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>League table</th>
</tr>
</thead>
<tbody>
<tr>
<td>id:long</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tournament table</th>
</tr>
</thead>
<tbody>
<tr>
<td>id:long</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Restructuring Activities

- Realizing associations
- Revisiting inheritance to increase reuse
- Revising inheritance to remove implementation dependencies
Realizing Associations

• 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
Unidirectional 1-to-1 Association

Object design model before transformation

```
ZoomInAction

MapArea
```

Object design model after transformation

```
ZoomInAction

MapArea
- zoomIn:ZoomInAction
+ getZoomInAction()
+ setZoomInAction(action)
```
Bidirectional 1-to-1 Association

Object design model before transformation

ZoomInAction - 1
MapArea - 1

Object design model after transformation

<table>
<thead>
<tr>
<th>ZoomInAction</th>
<th>MapArea</th>
</tr>
</thead>
<tbody>
<tr>
<td>-targetMap:MapArea</td>
<td>-zoomIn:ZoomInAction</td>
</tr>
<tr>
<td>+getTargetMap()</td>
<td>+getZoomInAction()</td>
</tr>
<tr>
<td>+setTargetMap(map)</td>
<td>+setZoomInAction(action)</td>
</tr>
</tbody>
</table>
1-to-Many Association

**Object design model before transformation**

Layer

1

LayerElement

**Object design model after transformation**

Layer

- `layerElements:Set`
  + `elements()`
  + `addElement(le)`
  + `removeElement(le)`

LayerElement

- `containedIn:Layer`
  + `getLayer()`
  + `setLayer(l)`

© 2007 Bernd Bruegge

Introduction into Software Engineering  Summer 2007
## Qualification

### Object design model before transformation

| Scenario | simname | 0..1 | SimulationRun |

### Object design model after transformation

<table>
<thead>
<tr>
<th>Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>- runs: HashTable</td>
</tr>
<tr>
<td>+ elements()</td>
</tr>
<tr>
<td>+ addRun(simname,sr:SimulationRun)</td>
</tr>
<tr>
<td>+ removeRun(simname,sr:SimulationRun)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SimulationRun</th>
</tr>
</thead>
<tbody>
<tr>
<td>- scenarios: Vector</td>
</tr>
<tr>
<td>+ elements()</td>
</tr>
<tr>
<td>+ addScenario(s:Scenario)</td>
</tr>
<tr>
<td>+ removeScenario(s:Scenario)</td>
</tr>
</tbody>
</table>
Increase Inheritance

• Rearrange and adjust classes and operations to prepare for inheritance

• Abstract common behavior out of groups 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.

• Be prepared to change a subsystem (collection of classes) into a superclass in an inheritance hierarchy.
Building a super class from several classes

- Prepare for inheritance. All operations must have the same signature but often the signatures do not match
- Abstract out the common behavior (set of operations with same signature) and create a superclass out of it.
- Superclasses are desirable. They
  - increase modularity, extensibility and reusability
  - improve configuration management
- Turn the superclass into an abstract interface if possible
  - Use Bridge pattern
Object Design Areas

1. Service specification
   • Describes precisely each class interface

2. Component selection
   • Identify off-the-shelf components and additional solution objects

3. Object model restructuring
   • Transforms the object design model to improve its understandability and extensibility

4. Object model optimization
   • Transforms the object design model to address performance criteria such as response time or memory utilization.
Design Optimizations

• Design optimizations are an important part of the object design phase:
  • The requirements analysis model is semantically correct but often too inefficient if directly implemented.

• Optimization activities during object design:
  1. Add redundant associations to minimize access cost
  2. Rearrange computations for greater efficiency
  3. Store derived attributes to save computation time

• As an object designer you must strike a balance between efficiency and clarity.
  • Optimizations will make your models more obscure
Design Optimization Activities

1. Add redundant associations:
   • What are the most frequent operations? (Sensor data lookup?)
   • How often is the operation called? (30 times a month, every 50 milliseconds)

2. Rearrange execution order
   • Eliminate dead paths as early as possible (Use knowledge of distributions, frequency of path traversals)
   • Narrow search as soon as possible
   • Check if execution order of loop should be reversed

3. Turn classes into attributes
Implement Application domain classes

• To collapse or not collapse: Attribute or association?

• Object design choices:
  • Implement entity as embedded attribute
  • Implement entity as separate class with associations to other classes

• Associations are more flexible than attributes but often introduce unnecessary indirection.

• Abbott's textual analysis rules

• Every student receives a number at the first day in the university.
Optimization Activities: Collapsing Objects

![Diagram showing the relationship between a Student and a Matrikelnumber with an ID: String property.]

The diagram illustrates the process of collapsing objects in optimization activities, specifically focusing on the relationship between a Student and their Matrikelnumber, with an ID: String property. This visualization helps in understanding the optimization techniques applied in software engineering.
To Collapse or not to Collapse?

• Collapse a class into an attribute if the only operations defined on the attributes are Set() and Get().
Design Optimizations (continued)

Store derived attributes

- Example: Define new classes to store information locally (database cache)

- Problem with derived attributes:
  - Derived attributes must be updated when base values change.
  - There are 3 ways to deal with the update problem:
    - Explicit code: Implementor determines affected derived attributes (push)
    - Periodic computation: Recompute derived attribute occasionally (pull)
    - Active value: An attribute can designate set of dependent values which are automatically updated when active value is changed (notification, data trigger)
Optimization Activities: Delaying Complex Computations

Image

<table>
<thead>
<tr>
<th>filename: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>data: byte[]</td>
</tr>
<tr>
<td>width()</td>
</tr>
<tr>
<td>height()</td>
</tr>
<tr>
<td>paint()</td>
</tr>
</tbody>
</table>

RealImage

<table>
<thead>
<tr>
<th>filename: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>data: byte[]</td>
</tr>
<tr>
<td>width()</td>
</tr>
<tr>
<td>height()</td>
</tr>
<tr>
<td>paint()</td>
</tr>
</tbody>
</table>

ImageProxy

<table>
<thead>
<tr>
<th>filename: String</th>
</tr>
</thead>
<tbody>
<tr>
<td>width()</td>
</tr>
<tr>
<td>height()</td>
</tr>
<tr>
<td>paint()</td>
</tr>
</tbody>
</table>

image

1

0..1
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 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.
Generalization: Finding the super class

- You need to prepare or modify your classes for generalization.
- All operations must have the same signature but often the signatures do not match.
- Superclasses are desirable. They
  - increase modularity, extensibility and reusability
  - improve configuration management
- Many design patterns use superclasses
  - Try to retrofit an existing model to allow the use of a design pattern
Implement 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 treated like attributes in the classes and translate to references
  • 1-to-many association:
    • Always Translate into a Vector
  • Qualified association:
    • Always translate into Hash table