Section 5
Implementation

1. Concepts
2. Mapping to collections
3. Mapping to storage

Implementation Outcomes

- Learning outcomes
  - understand strategies for mapping models to code
  - understand strategies for mapping models to persistent storage

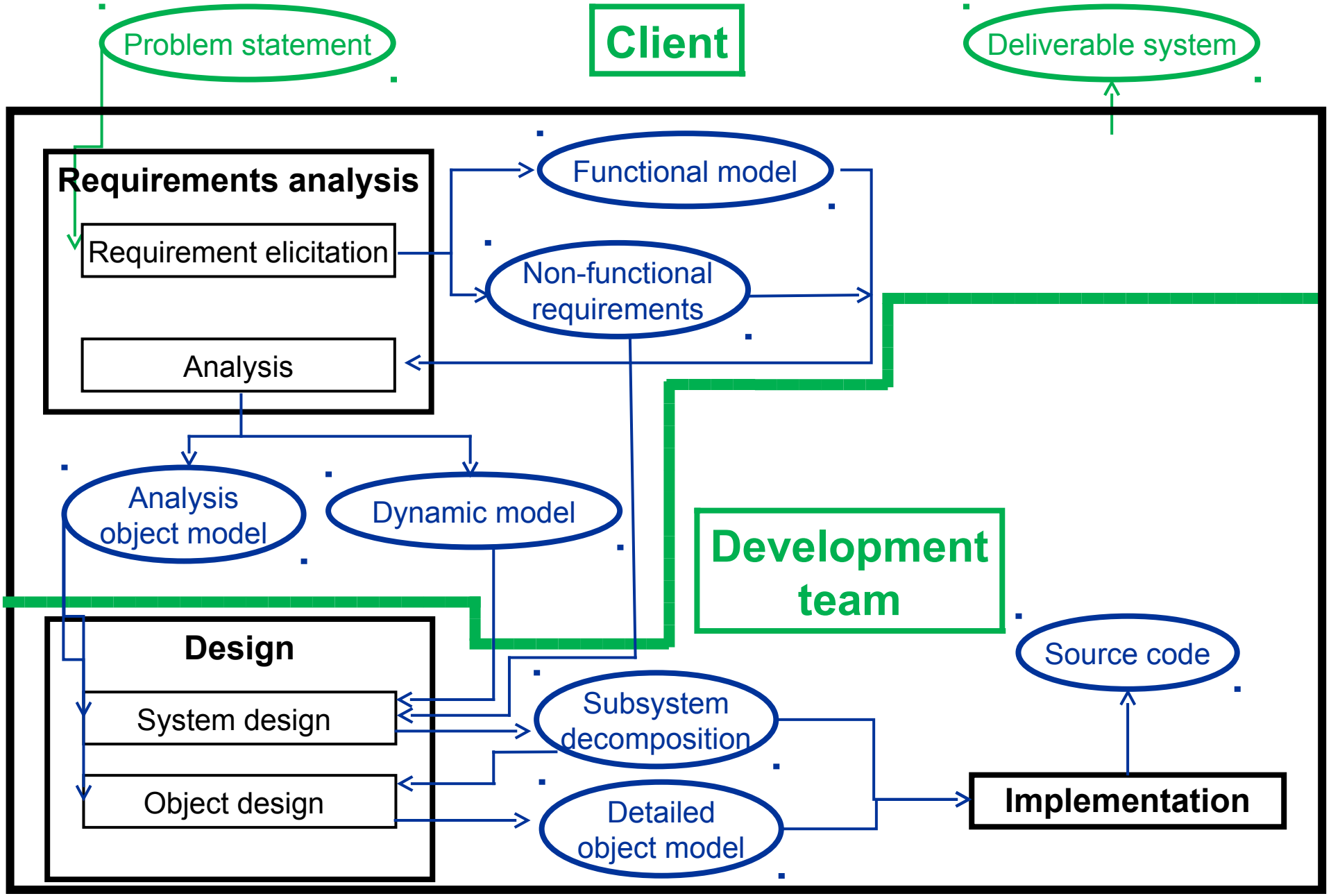
Section 5.1
Implementation Concepts

1. Overview
2. Model transformation
3. Optimizing the object model
4. Mapping contracts

5.1.1 Overview

- Input to implementation
  - subsystem decomposition
  - detailed object model
- Output from implementation
  - source code

Implementation Work Products



Crunch Mode

- Implementation is where things start to go wrong...

- Common problems
  - integration of subsystems that were developed by different teams
    - different handling of contract violations
    - undocumented changes to API
    - undocumented changes to classes and persistent data
  - delivery pressure
  - improvisations and workarounds

- These add up to code that doesn't match the design

Dealing With Crunch Mode

- Crunch mode:
  - "The way one works during crunch time. In an effort to make up for schedule slippage and meet a deadline, workers are required to make sacrifices including (but not limited to) sleep, nutrition, social life, hygiene, and product quality"

    -- www.urbandictionary.com

5.1.2 Model Transformation

- Model transformation overview
- Refactoring
- Forward engineering
- Reverse engineering
- Transformation principles

Model Transformation Overview

- What is model transformation?
  - changes that are applied to an existing object model
  - this results in a new object model

- Goal
  - simplify
  - optimize
  - get closer to meeting requirements

- Example
  - add/remove/rename classes, attributes, operations

Model Transformation Overview (cont.)

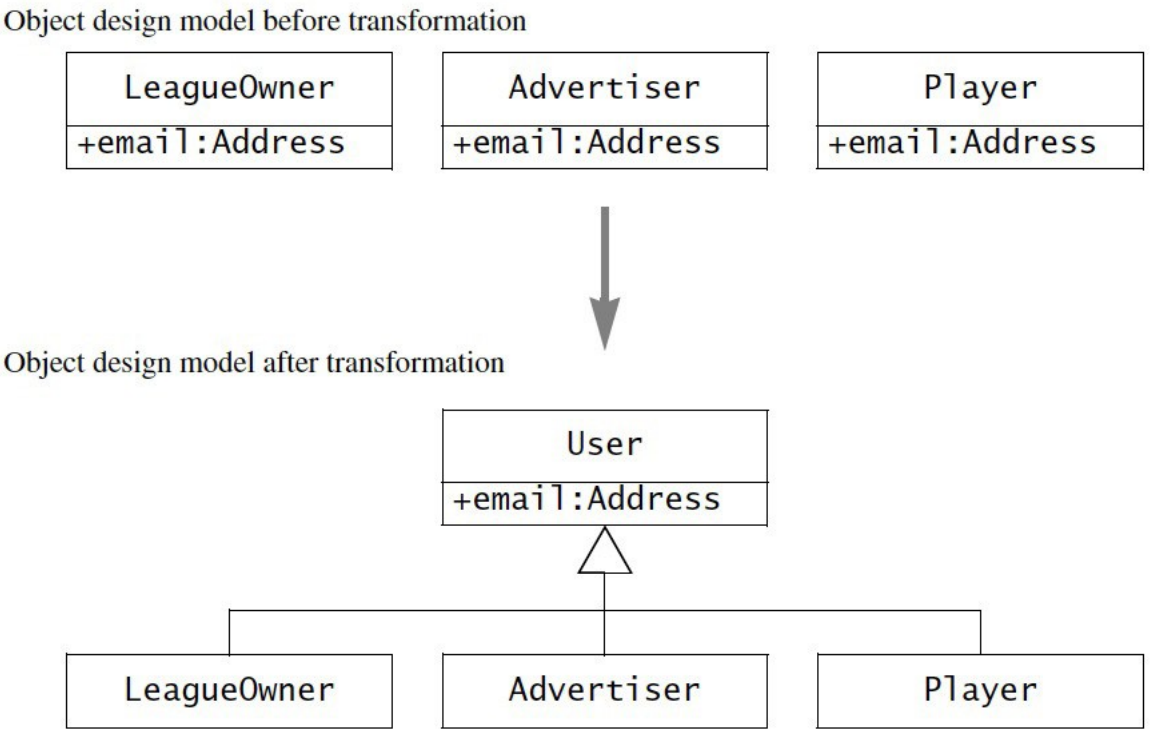


Figure 10-2 An example of an object model transformation. A redundant attribute can be eliminated by creating a superclass.

Model Transformation Overview (cont.)

- Goal of model transformations
  - improving one aspect of a model while preserving all its other properties

- Characteristics of model transformations
  - they must be localized
  - they must affect a small number of classes, attributes, operations
  - they must be executed in a series of small steps
  - they can occur anytime during:
    - object design
    - implementation

Model Transformation Overview (cont.)

- Optimizing class model
  - focus on performance requirements
  - reduce multiplicity of associations
  - add redundant associations for efficiency
  - add derived attributes

- Realizing associations
  - map associations to source code constructs
    - references
    - collections of references

Model Transformation Overview (cont.)

- Mapping contracts to exceptions
  - describe behaviour of operations when contracts are broken
    - where/when exceptions are raised
    - where exceptions are handled (at what layer in the software)
- Mapping class model to storage schema
  - define how class model relates to selected storage schema

Model Transformation Overview (cont.)

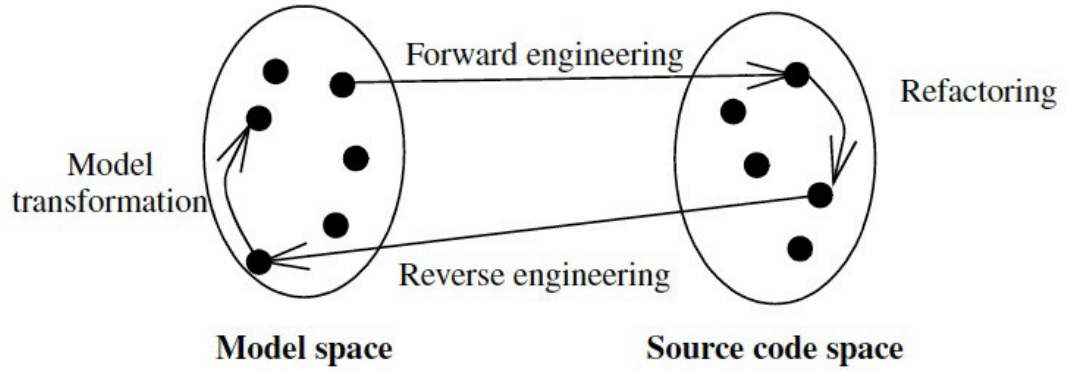


Figure 10-1 The four types of transformations described in this chapter: model transformations, refactorings, forward engineering, and reverse engineering.

Refactoring

- Goal of refactoring
  - improve the design of the system
- Characteristics of refactoring
  - it's applied to the source code
  - it improves readability or modifiability without changing behaviour
  - it is performed in small, incremental steps, interleaved with testing
  - it must focus on one attribute or operation at a time
- Example
  - generalizing a common attribute

Refactoring (cont.)

Before refactoring

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

After refactoring

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

Figure 10-3 Applying the *Pull Up Field* refactoring.

Refactoring (cont.)

Before refactoring

```
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;
    //...
    }
}
```

After refactoring

```
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);
    //...
    }
}
```

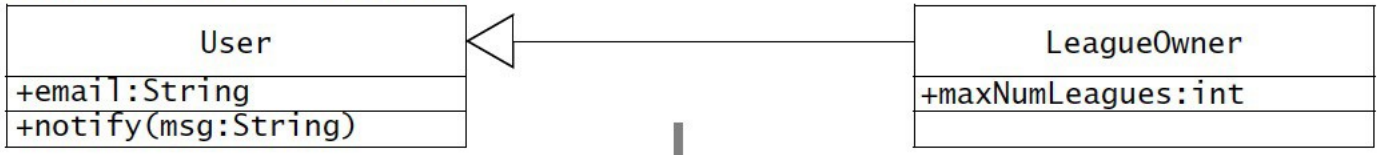
Figure 10-4 Applying the *Pull Up Constructor Body* refactoring.

Forward Engineering

- What is forward engineering?
  - writing the code
- Goal
  - maintain correspondence between object design model and code
  - reduce the number of implementation errors
- Characteristics of forward engineering
  - it is applied to a set of model elements
  - it results in a set of corresponding source code statements
    - class definition
    - language expression
    - database schema

Forward Engineering (cont.)

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 */
}
```

Figure 10-5 Realization of the `User` and `LeagueOwner` classes (UML class diagram and Java excerpts). In this transformation, the public visibility of `email` and `maxNumLeagues` denotes that the methods for getting and setting their values are public. The actual fields representing these attributes are private.

Reverse Engineering

- What is reverse engineering?
  - inferring the model from the code
- Goal
  - recreate the model for an existing, already implemented system
- Characteristics of reverse engineering
  - it is applied to a set of source code elements
  - it results in a set of model elements
  - it is the inverse transformation of forward engineering

Transformation Principles

- Overall approach
  - improve the design of the system with respect to some criterion
  - not introduce new errors

Transformation Principles (cont.)

- Principles
  - each transformation must address a single criterion
    - one transformation to meet one design goal
  - each transformation must be local
    - only a few classes or operations at once
    - changes to many subsystems are an *architectural change*
      - not a transformation
  - each transformation must be applied in isolation of other changes
    - one transformation at a time
  - each transformation must be followed by a validation step
    - validate each transformation after it is made
    - update appropriate models and documents

5.1.3 Optimizing the Object Model

- Optimizing access paths
  - repeated association traversals
    - identify frequent operations that require multiple association traversal
    - these should have direct connections instead
    - results in redundant connections, but may improve bottlenecks
  - "many" multiplicity associations
    - replace with "one" multiplicity *qualified* association
      - uses keys or indexing into objects on the "many" side

Optimizing the Object Model (cont.)

- Optimizing access paths (cont.)
  - misplaced attributes
    - for attributes involved in only get/set operations, fold into calling class
    - may result in fewer classes
- Result
  - selected redundant associations
  - fewer inefficient many-to-many associations
  - fewer classes

Optimizing the Object Model (cont.)

- Collapsing objects
  - objects may be replaced by attributes
  - special behaviour may have to be moved

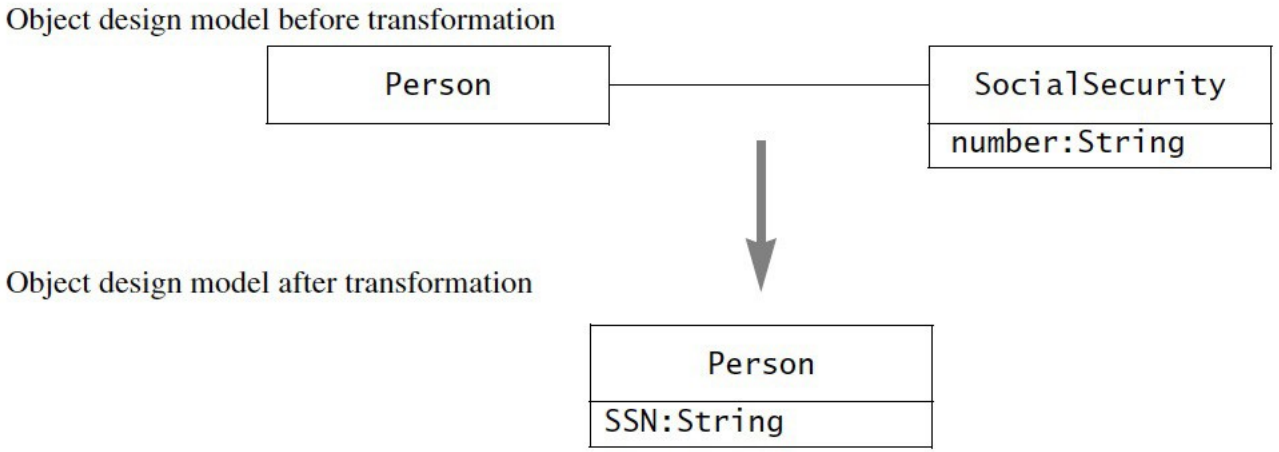


Figure 10-6 Collapsing an object without interesting behavior into an attribute (UML class diagram).

Optimizing the Object Model (cont.)

- Delaying expensive computations
  - if some objects are expensive to create, wait until they are needed

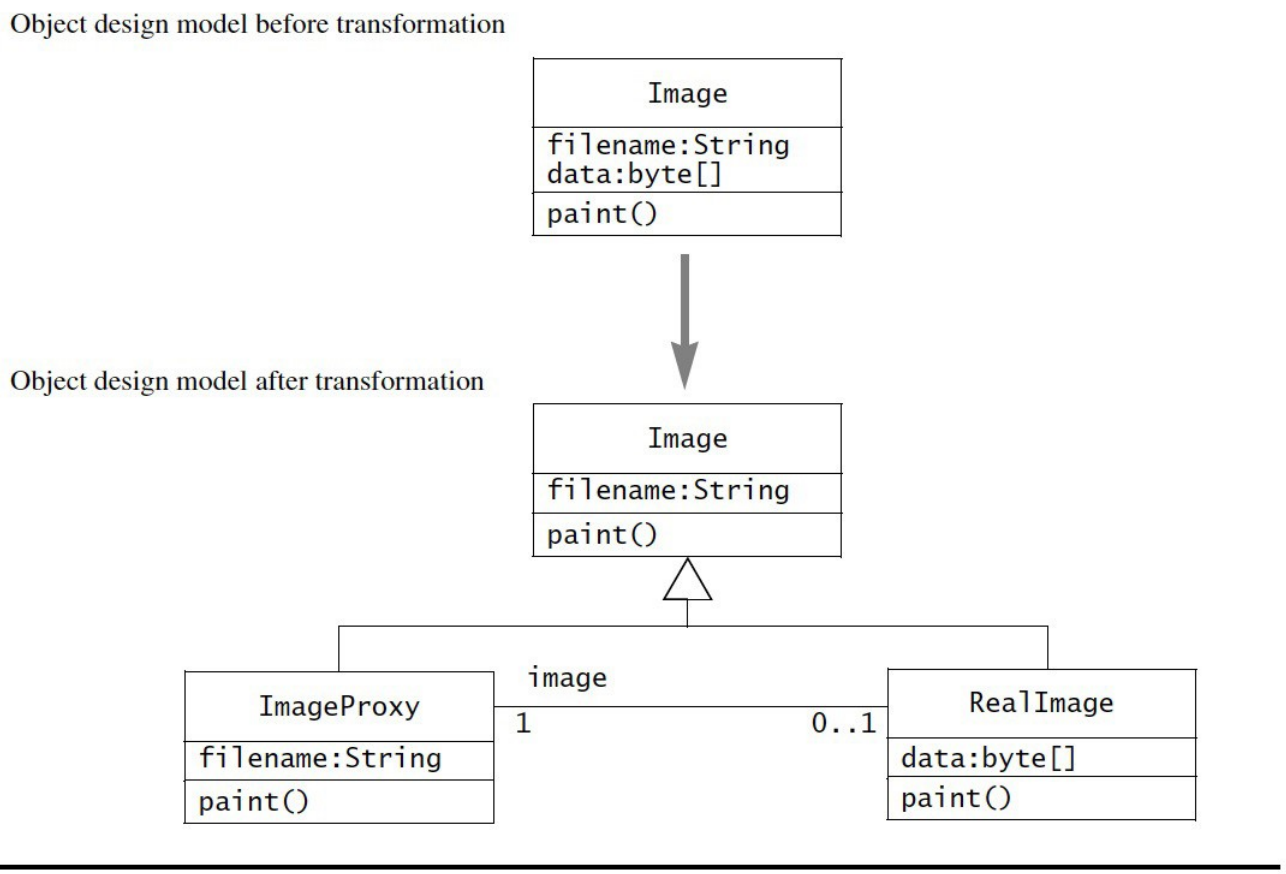


Figure 10-7 Delaying expensive computations to transform the object design model using a Proxy design pattern (UML class diagram).

Optimizing the Object Model (cont.)

- Caching the results of expensive operations

- Caching is suitable for:
  - frequently called operations
  - operations whose internal values seldom change

- Approach
  - internal values can be cached in private attributes
  - involves space/time trade-off

5.1.4 Mapping Contracts

- How are contracts mapped?
  - use exception handling to deal with contract violations
  - remember **`try-throw-catch`** in C++ and Java?

- Approach
  - it's easy to overdo this
  - if we check every precondition, postcondition, and invariant
    - it's too much work
    - we may introduce errors or mask existing bugs
    - code may get very convoluted
    - computational performance may take a hit

Mapping Contracts (cont.)

- Heuristics
  - don’t check for postconditions and invariants
    - this is usually redundant
  - focus on the system interfaces
    - check the public operations
    - don’t bother checking private or protected operations
  - focus on long-life components
    - pay special attention to code that is most likely to be reused
  - reuse constraint checking code
    - operations with similar preconditions can use encapsulated code
    - exception classes can be shared