

Mapping Models to Code

CSE 308: Software Engineering

Where We've Been

- [Requirements Elicitation
- [Analysis
- [System Design
- [Object Design

What's Left?

- [Implementation

 - mapping models to code

- [Testing

- [Rationale Management

- [Development methodologies

Mapping Models to Code

- [At this point in the development process, we confront integration problems
 - ex. undocumented parameters, inconsistent handling of contract violations
- [We apply a series of transformations to our model to avoid these kinds of problems

Transformation Types

- [Optimizing the class model
- [Mapping associations to collections
- [Mapping operation contracts to exceptions
- [Mapping the class model to a storage schema

Mapping Overview

- [Transformation: operation that aims to improve one aspect of a model while preserving all of its other properties
- [Transformations are usually localized
- [Transformations usually consist of a series of small steps

Mapping Concepts

- [Model transformation — operates on object models
- [Refactoring — operates on source code
- [Forward engineering — produces new source code from an object model
- [Reverse engineering — produces a model that corresponds to source code

Model Transformations

- [Goal is to simplify or optimize an object model
- [Produces another object model
- [May add/remove/rename classes, operations, etc.
- [ex. Analysis phase organizes objects into inheritance hierarchies and eliminates redundancy
- [Development as a whole is a series of model transformations

Refactoring

- [Source code transformation
 - improves readability and/or modifiability
 - does not change system behavior
- [Performed in small steps accompanied by tests
- [Types: Pull Up Field, Pull Up Constructor, Pull Up Method

Forward Engineering

- [Generates source code that corresponds to a set of model elements
- [Tends to produce identical code, making it easier to spot locations where this has been done
- [Consistent approach reduces errors

Reverse Engineering

- [Generates a set of model elements based on source code
- [Used to recreate models when source code is all that's available, or when source code and models are out of sync
- [Does not necessarily recreate the same model, because forward engineering may lose information
 - only useful as an approximation of the original model

Transformation Principles

- [To avoid introducing new errors, transformations should follow four principles:
 - each transformation must address a single criterion
 - each transformation must be local
 - each transformation must be applied in isolation
 - each transformation should have a validation step

Mapping Activities

Optimizing The Design Model

- [Direct translation of an analysis model into source code is often inefficient
 - analysis model focuses on function, not design goals
- [Optimization requires a balance between efficiency and clarity (system complexity)

Optimizing Access Paths

- [Repeated association traversals

- frequent operations should not require traversal of multiple associations

- use a direct connection instead

- ["Many" associations

- use qualified associations to reduce "many"-multiplicity associations to "one"-multiplicity associations

Optimizing Access Paths

- [Misplaced attributes

- may have classes with no interesting behavior
- fold attributes into the calling class
- remove unneeded classes

Collapsing Objects

- [Classes with only a few behaviors/attributes can be collapsed into attributes themselves
- [Refactoring procedure:
 - declare source's public fields/methods in absorbing class
 - Change references/class name
 - Delete source class

Delaying Expensive Computations

- [Delay object creation until absolutely necessary
 - Proxy design pattern
 - takes place of an expensive class
- [Cache the results of expensive computations as private attributes

Mapping Associations to Collections

- [Associations denote bidirectional links between objects
- [O-O languages only provide one-way references
- [During object design, we use references to implement associations

Mapping Associations

- [Unidirectional one-to-one associations

- use a simple reference to represent this association

- [Bidirectional one-to-one associations

- each object has a reference to the other

- ex. using "this" as constructor argument

Mapping Associations

- [One-to-many associations

- use a collection of references

- “many” side collection depends on association constraints

- [Many-to-many associations

- both end classes have collections of references

- need to ensure that collections are consistent

Mapping Associations

- [Qualified associations

- reduce multiplicity of a “many” side
- use a *Map* to represent the qualified end
- qualifier is passed as a parameter

Mapping Associations

- [Associations classes

- holds attributes and operations of an association

- Transform association class into an object, then convert binary association to reference attributes

Mapping Contracts to Exceptions

- [Some O-O languages can automatically raise exceptions when constraints are violated
 - Java is not one of these languages
- [Java can use the “throw” keyword to explicitly raise exceptions when contracts are violated
 - throw unwinds the call stack to find a catch block

One Approach

- [Test each precondition at the top of the method, raising unique exceptions if violations are detected
- [Test each postcondition before the method ends, again raising unique exceptions for violations
 - Check invariants at the same time
- [Inherited pre/postconditions should be encapsulated into separate methods that can be called by subclasses

Problems With This Approach

- [Increased coding effort
- [Increased defect opportunity
- [Obfuscated code
- [Performance impact

- [Instead, we need to adopt a pragmatic approach

Exception-Mapping Heuristics

- [Omit checking code for postconditions and invariants
 - usually redundant, should be written by someone else
- [Focus on subsystem interfaces
 - omit checking code for private/protected methods
- [Focus on contracts for components with the longest life
- [Reuse constraint-checking code (via refactoring)

Mapping to Persistent Storage

- [OO programming languages usually lack an efficient way to store persistent objects
 - need to map persistent objects to some data structure
- [No such transformations are required for object-oriented databases
- [Here, we focus on mapping to relational databases

Basic Terminology

- [Schema: description of the data

- describes valid set of data records stored in the DB

- [Relational databases store data in tables (relations)

- tables consist of columns, one per attribute

- [Primary key: set of attributes used to uniquely identify data records

- [Foreign key: references the primary key of another table

firstName	login	email
"alice"	"am384"	" <u>am384@mail.org</u> "
"john"	"js289"	" <u>john@mail.de</u> "
"bob"	"bd"	" <u>bobd@mail.ch</u> "

name	login
"tictactoeNovice"	"am384"
"tictactoeExpert"	"am384"
"chessNovice"	"js289"

Classes and Attributes

- [Map each class to a table with the same name
 - table columns (attributes) map to class attributes
 - each data record corresponds to a class instance
- [When mapping attributes, look for corresponding DB types
- [Choosing a primary key:
 - use a set of class attributes, or
 - add a unique identifier attribute (more robust)

Mapping Associations

— [Buried associations

- used for one-to-one and one-to-many associations
- add a foreign key to the table on the “many” end

— [Separate tables

- used for many-to-many associations
- use separate two-column association table
- tradeoff between modifiability vs. added table overhead

Mapping Inheritance

- [Vertical mapping

- each class is represented by a table

- foreign keys link subclass tables to the superclass table

- [Horizontal mapping

- attribute columns are duplicated in subclass tables

Vertical Mapping

- [Superclass table has an additional attribute/column for a data record's subclass
- [Subclass has columns for all attributes
- [All tables share the primary key (object identifier)
- [To retrieve an object, start by examining superclass table, then query subclass table

id	name	...	role
56	zoe		LeagueOwner
79	john		Player

id	maxNumLeagues	...
56	12	

id	credits	...
79	126	

Horizontal Mapping

- [Pushes superclass attributes into subclasses
 - removes need for superclass table
 - each subclass table includes columns for its own attributes as well as the superclass attributes
- [Reduces lookup time, but makes it more difficult to add superclass attributes (each subclass must be modified)

Managing Implementation

Documenting Transformations

- [Transformations provide systematic recipes for improving specific aspects of the object design model
- [We need to document these transformations so that we can reapply them when the object design model/code changes
- [In particular, most useful transformations are NOT one-to-one mappings
 - information is lost in the transformation process

Preserving Consistency

- [For a given transformation, use the same tool
- [Keep contracts in the source code, not object design model
- [Use the same names for the same objects
- [Make transformations explicit
 - ex. use a coding conventions guide

Assigning Responsibilities

- [Core architect

- selects the transformations to be systematically applied

- [Architecture liaison

- documents contracts associated with subsystem interfaces

- [Developer

- follows conventions set by the core architect
- actually applies transformations

Next Time

- [Chapter 11: Testing
 - Testing concepts
 - Testing activities
 - Testing management