Class Design

Fabrizio Maria Maggi

Institute of Computer Science

(these slides are derived from the book "Object-oriented modeling and design with UML")

System Conception

- Is the genesis of an application
- A person who understands both business needs and technology thinks of an idea for an application
  - In a later stage developers explore the idea to devise possible solutions
- The purpose of system conception is to defer details and understand the big picture
  - Who is the application for?
  - What problems will it solve?
  - Where will it be used?
  - When is it needed?
  - Why is it needed?
  - How will it work?

Domain Analysis VS Application Analysis

- Domain Analysis
  - Focuses on real-world things whose semantics the application captures
    - Example: a flight is a real-world object that a flight reservation system must represent
  - Domain objects exist independently of any application and are meaningful to business experts
- Application Analysis
  - Addresses the aspects of the application that are visible to users
    - Example: a flight reservation screen is part of a flight reservation system
  - Application objects do not exist in the problem domain and are meaningful only in the context of an application
  - The application model does not prescribe the implementation of an application
    - It describes as a black–box how the application appears from the outside

Domain Analysis

- Domain Class Model
  - Find classes
  - Keep the right classes
  - Find associations
  - Keep the right associations
  - Find attributes
  - Keep the right attributes
  - Grouping classes into packages
- In general, domain analysis does not include interaction models

Application Analysis

- Application Interaction Model
  - Determine the system boundary
  - Find actors
  - Find use cases
  - Prepare normal scenarios
  - Prepare exception scenarios
  - Define sequence diagrams
- Application Class Model
  - Specify user interfaces
    - A user interface is an object that provides the user of a system with a coherent way to access its domain objects, commands and application options
  - Define boundary classes
    - A boundary class is a class that manages communications between a system and an external source
  - Determine controllers
    - A controller is a class that manages control within an application
  - Add operations
    - They are mainly derived from the interaction model

Overview of Class Design

- In general class design is a process to add details to the class diagrams defined in the analysis phase and making fine decisions
- You choose how to implement your classes considering
  - Minimization of execution time, memory and other cost measures
  - Choice of algorithms implementing methods
  - Braking complex operation ito simpler operations
- OO design is an iterative process
  - When one level of abstraction is complete you should design the next lower level of abstraction
  - For each level you may
    - add new operations, attributes, and classes
    - Revise relations between classes

The Essence of Design is…

- To build a bridge across the gap between:
  - Desired features
    - Use cases
    - Application commands
    - System operations
    - System services
  - Available resources
    - Operating system infrastructure
    - Class libraries
    - Previous applications

Creativity

- Design is a creative task
  - You can only be provided with guidelines
- The tools you have to fill the gap between resources and features are
  - Classes
  - Operations
  - Other UML constructs
- You may have to compromise
  - The final goal is not to choose the best for single decisions but to optimize the entire system

Realizing Use Cases

- Use Cases define the functionalities of a system but not how to realize them
  - One goal of the design phase is to choose among different possible realizations (finding a balance between advantages and disadvantages)
- Implement the required behavior is not sufficient
  - You should take into consideration performance, reliability, facilitating possible future enhancements
- Use Cases define system level operations
  - During design you invent new objects and new operations (at a lower level of abstraction) to provide this behavior
  - Again: Bridge the gap!

Responsibilities

- First step for realizing a use case is to list its responsibilities
- A responsibility is something that a us case must do to be implemented
- Example: Online theater ticket system
  - Use case: Making Reservation
  - Responsibilities:
    - Finding unoccupied seats for the desired show
    - Marking the seats as occupied
    - Obtaining payment from the customer
    - Arranging delivery of the tickets
    - Crediting payment to the proper account

Responsibilities

- Each use case has various responsibilities
  - Some responsibilities are in common for different use cases and can be reused
- Group responsibilities in clusters consisting of related responsibilities
  - Each cluster must be implemented by a single lower-level operation
  - Define an operation for each cluster
    - The operation should be general enough to be used in several different places of the current design
  - Assign the lower-level operations to classes
    - If there is no good class to hold an operation, introduce new lower-level classes

Designing Algorithms

- Formulate an algorithm for each operation
  - The analysis specification of an operation tells *what* the operation does, the algorithm shows *how* it is done
- To design an algorithm you have to:
  - Choose algorithms that minimize the cost of implementing operations
  - Select data structures appropriate to the algorithms
  - Define new internal classes and operations as necessary
  - Assign operations to appropriate classes

Choosing Algorithms

- Many operations are very simple: they just get or change attribute values into objects
- For more complex operations you can use pseudocode to define algorithms

```
Node::computeTransitiveClosure () returns NodeSet
   nodes:= createEmptySet;
   return self.TCloop (nodes);
Node::TCloop (nodes:NodeSet) returns NodeSet
   add self to nodes;
   for each edge in self.Edge
      for each node in edge.Node
         /* 2 nodes are associated with an edge */
         if node is not in nodes then node.TCloop(nodes);
         end if
      end for each node
   end for each edge
```

Figure 15.3 Pseudocode example. You can express difficult algorithms with pseudocode. The top method initiates computation and the bottom method recurses for nodes that are one edge away and have not been visited before.

Object-Oriented Modeling and Design with UML, Second Edition by Michael Blaha and James Rumbaugh. ISBN 0-13-1-015920-4. © 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Choosing Algorithms

- If efficiency is not an issue, use simple algorithms
- Try to improve performances only for operations that are a bottleneck
- Choose among alternative algorithms based on:
  - Computational complexity
    - How does processor time increase as a function of data structure size?
  - Ease of implementation and understandability
    - It's worth giving up some performance on noncritical operations if you can use simple algorithms (making a model easier to understand and easier to program)
  - Flexibility
    - Maintainability is crucial. A highly optimized algorithm often sacrifice ease to change

Designing Algorithms

- Choosing data structures
  - Data structures do not add information to the analysis model
  - Data structures organize information to permit efficient algorithms
  - Data structures include: Arrays, Lists, Trees, Sets, etc.
- Defining internal classes and operations
  - Expanding high level operation through algorithms may lead to create new (low-level) classes and operations to hold intermediate results

Designing Algorithms

- Assigning operations to classes
  - When only an object is involved in an operation, the object itself will perform the operation
  - When more than one object is involved in an operation, the operation will be performed by the object that play the lead role in the operation
    - Target of action
    - Query vs. update
    - Focal class
    - Analogy to real world

Recursing Downward

- Organize operations as layers
  - Operations in higher layers invoke operations in lower layers
- In general, the design process works top down
  - Start with the higher-level operations and proceed to define lower-level operations
  - Functionality Layers
    - High-level functionalities are decomposed into lesser operations
    - Implementation of the responsibilities
  - Mechanism Layers
    - Support mechanisms to make the system working
      - Store information, coordinate objects, sequence control, transmit information, perform computations, etc.

Refactoring

- The initial design always contains inconsistencies, redundancies, inefficiencies
- It is impossible to get a large correct design in one pass
- Refactoring is an essential part of any good engineering process
- It's not enough to deliver a functionality
  - If you expect to maintain a design, then you must keep the design clean, modular and understandable

Design Optimization: Provide efficient access paths

- Adjusting the structure of the class model to optimize frequent traversals

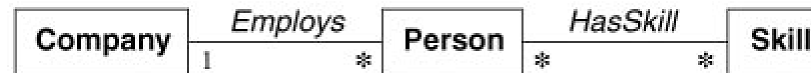


Figure 15.5 Analysis model for person skills. Derived data is undesirable during analysis because it does not add information.

Object-Oriented Modeling and Design with UML, Second Edition by Michael Blaha and James Rumbaugh. ISBN 0-13-1-015920-4. © 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

- Company.findSkill(*speakJapanese*)
  - Frequency of access
  - Fan out (e.g., 1000 employees with on average 10 skills)
  - Selectivity (e.g., if only 5 employees actually speak Japanese)

Design Optimization: Provide efficient access paths

- Frequency of access (High)
- Fan out (High)
- Selectivity (Low)

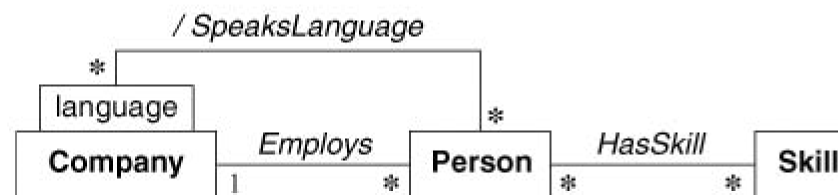


Figure 15.6 Design model for person skills. Derived data is acceptable during design for operations that are significant performance bottlenecks.

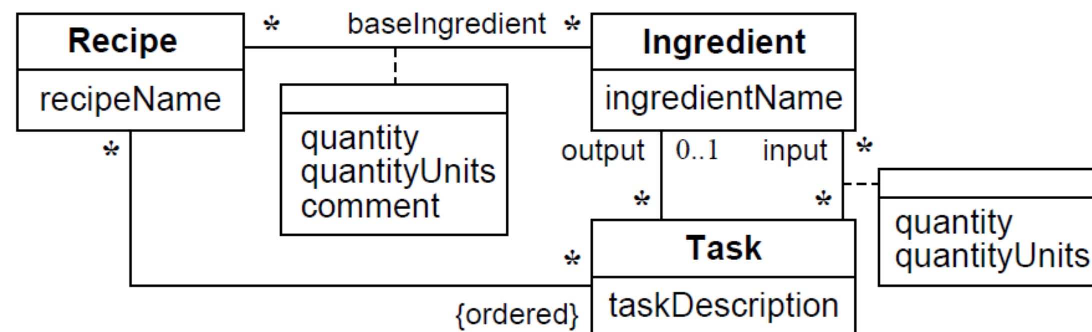
Object-Oriented Modeling and Design with UML, Second Edition by Michael Blaha and James Rumbaugh. ISBN 0-13-1-015920-4. © 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Design Optimization: Rearrange execution order for efficiency

- Optimizing the algorithm eliminating dead paths
  - Find employees who speak both French and Japanese
    - You know that there are less people who speak Japanese
    - What is the right testing order to compute the result?
- Saving derived values to avoid recomputation
  - Explicit update
    - The code for updating the base attributes also updates the derived ones
  - Periodic recomputation
    - You recompute all the derived attributes periodically
  - Active values
    - The derived attribute automatically monitors the base attributes to check if they change

Reification of Behavior

- Behavior written in code is rigid
- If you want to manipulate, pass store the behavior at runtime you should reify it
- Reification is the promotion of something that is not an object into an object
- Example (exercise 4.16, Chapter 4):



Adjustment of Inheritance

- Through the following steps:
  - Rearrange classes and operations to increase inheritance
  - Abstract common behavior out of groups of classes
  - Use delegation to share behavior when inheritance is semantically invalid

Rearrange classes and operations

- Sometimes several classes define the same operation that can be easily inherit from a common ancestor
- Often operations are similar but not identical. To cover them with a single inherited operation you have to find:
  - Operations with optional arguments
  - Operations that are special cases
  - Inconsistent names
  - Irrelevant operations

Abstracting Out Common Behavior

- If 2 classes repeat several operations and attributes they can be specializations of the same ancestor
- Abstracting out
  - Means to create a common (abstract) superclass for the shared features leaving only the specialized features in the subclasses

Using Delegation to Share Behavior

- Avoid inheritance of implementation
  - i.e., use inheritance to reuse code without taking into consideration the semantics of the classes
- Use delegation
  - Catching an operation and send it to a related object
- Example: Stack versus List
- Java?

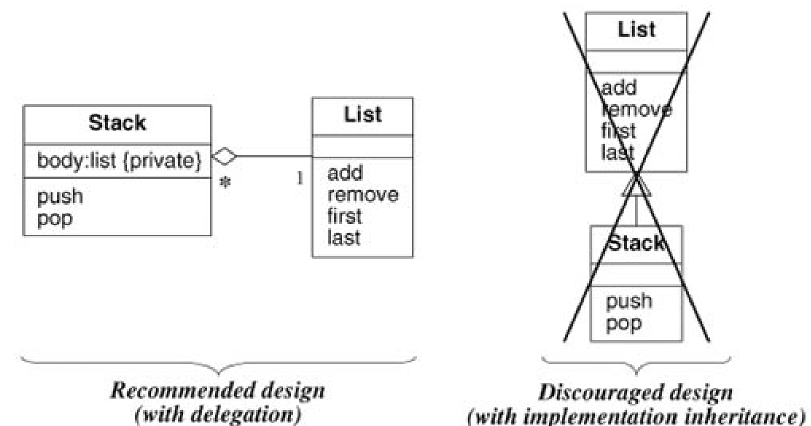


Figure 15.7 Alternative designs. Do not use implementation inheritance.

Object-Oriented Modeling and Design with UML, Second Edition by Michael Blaha and James Rumbaugh. ISBN 0-13-1-015920-4. © 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Organizing a Class Design

- Information hiding
  - Separating external specification from internal implementation
- Coherence of Entities
  - An entity (a class, an operation or a package) should have a single major theme
    - A method should do only 1 thing
    - A class should not serve many purposes at once
- Fine-Tuning Packages
  - The interface between two packages (the associations that relate classes in one package to classes in the other and operations that access classes across package boundaries) should be minimal and well defined