Classes – Domain, Design, and Implementation

Sudipto Ghosh
CS314
Reference: Modeling with Classes
Chapter 11 in custom textbook (pages 394-434)
Chapter 5 in Lethbridge (pages 169-209)

UML – Unified Modeling Language

- The objective of UML is to assist in software development
  - It is not a methodology
- A model should
  - use a standard notation
  - be understandable by clients and users
  - lead software engineers to have insights about the system
  - provide abstraction
- Models are used:
  - to help create designs
  - to permit analysis and review of those designs.
  - as the core documentation describing the system.
UML Class Diagrams

- **Domain models**
  - Represent the concepts in the problem domain, and their relationships
  - Also called conceptual models or analysis class models

- **Design class models**
  - Represent the classes in the solution

- **Implementation class model**
  - Represent the actual classes used in the program
  - Can be obtained using reverse engineering tools

Increasing detail

Motivation

- Reduce representational gap between mental and software models
UML Class Diagrams

The main symbols shown on class diagrams are:

- **Classes**
  - represent the types of data themselves

- **Associations**
  - represent linkages between instances of classes

- **Attributes**
  - are simple data found in classes and their instances

- **Operations (not used in domain models)**
  - represent the functions performed by the classes and their instances

- **Generalizations**
  - group classes into inheritance hierarchies

Domain models

- Also called conceptual model in the past
  - Or, requirements class diagram

- Contains
  - Class names
  - Class attributes
  - Associations
  - **NO operations**

- Avoid “analysis paralysis” – don’t waste time on a too thorough model
  - Work iteratively
Partial domain model (POS system) – a visual dictionary

What does a domain model show?

SalesDatabase and print() are software artifacts and not part of the domain model. 
Sale is a software class and not part of the domain model. 
DateTime is a concept in the domain of interest and not a software class.
What are conceptual classes?

A sale represents the event of a purchase transaction. It has a date and time.

Example

UP Domain Model
Stakeholder's view of the noteworthy concepts in the domain.

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former inspired the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

UP Design Model
The object-oriented developer has taken inspiration from the real world domain in creating software classes. Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Domain Modeling
Creating a domain model

- Find conceptual classes
  - Reuse or modify existing models
  - Use a category list
  - Identify noun phrases
- Draw concepts as classes in a UML diagram
- Add associations and attributes

Classes

A class is simply represented as a box with the name of the class inside
- The diagram may also show the attributes
A simple technique for discovering domain classes

- Look at a source material such as a description of requirements
- Extract the nouns and noun phrases
- Eliminate nouns that:
  - are redundant
  - represent instances
  - are vague or highly general
  - not needed in the application
- Pay attention to classes in a domain model that represent *types of users* or other actors

Identifying associations and attributes

- Start with classes you think are most **central** and important
- Decide on the clear and obvious data it must contain and its relationships to other classes.
- Work outwards towards the classes that are less important.
- Avoid adding many associations and attributes to a class
  - A system is simpler if it manipulates less information
Tips about identifying and specifying valid associations

☐ An association should exist if a class
  ■ possesses
  ■ controls
  ■ is connected to
  ■ is related to
  ■ is a part of
  ■ has as parts
  ■ is a member of, or
  ■ has as members some other class in your model

☐ Specify the multiplicity at both ends
☐ Label it clearly.

Actions versus associations

☐ A common mistake is to represent actions as if they were associations

Bad, due to the use of associations that are actions
Identifying attributes

- Look for information that must be maintained about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
  - E.g. string, number

Tips about identifying and specifying valid attributes

- It is not good to have many duplicate attributes
- If a subset of a class’s attributes form a coherent group, then create a distinct class containing these attributes
An example (attributes and associations)

<table>
<thead>
<tr>
<th>Passenger</th>
<th>Employee</th>
<th>RegularFlight</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
<td>time</td>
</tr>
<tr>
<td>number</td>
<td>employeeNumber</td>
<td>flightNumber</td>
</tr>
<tr>
<td></td>
<td>jobFunction</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Booking</th>
<th>SpecificFlight</th>
</tr>
</thead>
<tbody>
<tr>
<td>seatNumber</td>
<td>date</td>
</tr>
</tbody>
</table>

Associations and Multiplicity

An association is used to show how two classes are related to each other.

- Symbols indicating *multiplicity* are shown at each end of the association.

![Diagram of associations and multiplicity](image)

- Employee : 0..1
- Secretary : 1..*
- Manager : 1..*
- Company : 0..*
- BoardOfDirectors : 0..1
- Office : 0..1
- Employee : 0..1
- BoardOfDirectors : 0..1
- Person : 0..3..8
Labeling associations

- Each association can be labelled, to make explicit the nature of the association

- supervisor

- worksFor

- 0..1

- allocatedTo

- 0..3..8

- boardMember

Many-to-one associations

- A company has many employees,
- An employee can only work for one company.
  - This company will not store data about the moonlighting activities of employees!
- A company can have zero employees
  - E.g. a ‘shell’ company
- It is not possible to be an employee unless you work for a company
**Many-to-many associations**

- A secretary can work for many managers
- A manager can have many secretaries
- Secretaries can work in pools
- Managers can have a group of secretaries
- Some managers might have zero secretaries.
- Is it possible for a secretary to have, perhaps temporarily, zero managers?

<table>
<thead>
<tr>
<th>Secretary (1..*)</th>
<th>Manager (1..*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>supervisor</td>
<td></td>
</tr>
</tbody>
</table>

**One-to-one associations**

- For each company, there is exactly one board of directors
- A board is the board of only one company
- A company must always have a board
- A board must always be of some company
Common mistakes

- Avoid unnecessary one-to-one associations

Avoid this

<table>
<thead>
<tr>
<th>Person</th>
<th>PersonInfo</th>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>address</td>
<td>name</td>
</tr>
<tr>
<td></td>
<td>email</td>
<td>address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>email</td>
</tr>
<tr>
<td></td>
<td></td>
<td>birthdate</td>
</tr>
</tbody>
</table>

do this

A more complex example

- A booking is always for exactly one passenger
  - no booking with zero passengers
  - a booking could never involve more than one passenger.
- A Passenger can have any number of Bookings
  - a passenger could have no bookings at all
  - a passenger could have more than one booking
Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes.
- The following are equivalent:

  ![Diagram of association classes](image)

Reflexive associations

- It is possible for an association to connect a class to itself.

  ![Diagram of reflexive associations](image)
Generalization

- Specializing a superclass into two or more subclasses
  - The discriminator is a label that describes the criteria used in the specialization

Initial POS domain model

- Register
- Item
- Store
- Sale
- Sales
- LineItem
- Cashier
- Customer
- Ledger
- Cash Payment
- Product Catalog
- Product Description
Initial Monopoly domain model

Guidelines

- Maintain model in a tool or not?
- Include “Receipt” in POS domain model? (Discuss)
- Use domain terms
  - How about modeling the software world? (Telecommunication)
Should Receipts be included?

- Receipt = record of sale and payment
- Reason to exclude:
  - All the information in receipt is derived from other sources
- Reason to include:
  - Special role in terms of business rules: it confers the bearer the right to return their bought items.
- If “Return Items” is not included in this development cycle, then exclude Receipt

Instance Diagrams

- A *link* is an instance of an association
  - In the same way that we say an object is an instance of a class
More Advanced Features: Aggregation

- Aggregations are special associations that represent ‘part-whole’ relationships.
  - The ‘whole’ side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named *isPartOf*

When to use an aggregation

- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that
    - the parts *are part of* the aggregate
    - or the aggregate *is composed of* the parts
  - When something owns or controls the aggregate, then they also own or control the parts
Composition

- A composition is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well

- Two alternatives for addresses

Aggregation hierarchy
Domain model vs System model

- The *system domain model* omits many classes that are needed to build a complete system
  - Can contain less than half the classes of the system.
  - Should be developed to be used independently of particular sets of
    - user interface classes
    - architectural classes
- The complete *system model* includes
  - The system domain model
  - User interface classes
  - Architectural classes
  - Utility classes

Interfaces

An interface describes a portion of the visible *behaviour* of a set of objects.

- An *interface* is similar to a class, except it lacks instance variables and implemented methods
Suggested sequence of activities for design classes

- Identify a first set of candidate classes
- Add associations and attributes
- Find generalizations
- List the main responsibilities of each class
- Decide on specific operations
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces
  - Apply design patterns (we will see them later)

Identifying design classes

- When developing a domain model you tend to discover classes
- When you work on the user interface or the system architecture, you tend to invent classes
  - Needed to solve a particular design problem
  - (Inventing may also occur when creating a domain model)
- Reuse should always be a concern
  - Frameworks
  - System extensions
  - Similar systems