Object Oriented Concepts with UML 203105207

Prof. Shaleen Shukla, Assistant Professor
Information Technology Department

CHAPTER-6

Class design

Topics to be Covered

- Overview of class design
- Designing Algorithms Recursing Downward
- Refactoring
- design optimization
- Adjustment of Inheritance
- Reification of Behavior
- Design pattern: introduction and classification
- case study of model view controller (MVC).

INTRODUCTION

- The analysis phase determines what implementation must do
- The system design phase determines the plan of attack
- The purpose of class design is
- To complete the definitions of the classes and associations and

  choose algorithms for operations

Overview of class design

- Bridge the gap from high-level requirements to low-level services.
- Realize use cases with operations.
- Formulate an algorithm for each operation.
- Recurse downward to design operations that support higher-level operations.
- Refactor the model for a cleaner design

Overview of class design (conti..)

- Refactor the model for a cleaner design
- Optimize access paths to data
- Rectify behavior that must be manipulated
- Adjust class structure to increase inheritance
- Organize classes and associations

Bridging the Gap

- There is a set of features that you want your system to achieve.
- You have a set of available resources.
- Think of the distance between them as a gap.
- Your job is to build a bridge across the gap.

  Use cases

  Application commands

  System operations

  System services

Bridging the Gap

- Available resources
  - Operating system infrastructure
  - Class libraries
  - Previous applications
- If you directly construct each resources, you are done
- But usually it is not easy.
- Ex. You want to build a web-based ordering system
  - System services

Bridging the Gap

- You cannot readily build it from spreadsheet or programming language.
- You must invent some intermediate elements.
- The intermediate elements may be operations, classes or other UML constructs

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.

Realizing Use cases

- 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 use 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

Responsibilities

Arranging delivery of the tickets

Crediting payment to the proper account

- 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

Responsibilities

- 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 pseudo code to define 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:

Choosing Algorithms

- 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 Algorithm

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

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
  - Receiver 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 layer
- In general, the design process works top down
  - Start with the higher-level operations and proceed to define
- Two ways of recursing downward
- Functionality Layers
  - High-level functionalities are decomposed into lesser operations
  - Implementation of the responsibilities

Recursing Downward

- Mechanism Layers
  - Support mechanisms to make the system working.
  - 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
- Refactoring is changing the internal structure of the software to improve its design without altering its external functionalities
- 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

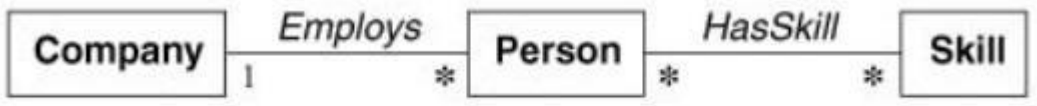


Figure 15.5 Analysis model for person skills. Derived data is undesirable during analysis because it does not add information.

- 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

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

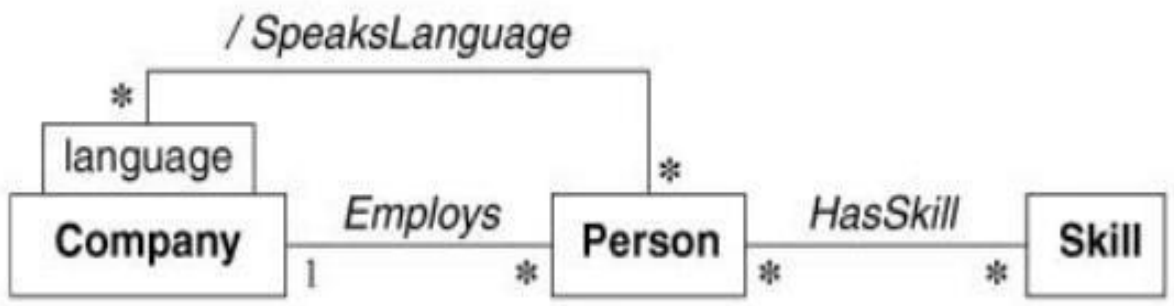


Figure 15.6 Design model for person skills. Derived data is acceptable during design for operations that are significant performance bottlenecks.

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?

Design Optimization: Saving derived values to avoider computation

- Sometimes it is helpful to define new classes to cache attributes and avoid recomputation.
- You must update cache if any of the objects on which it depends are changed
- Three ways to handle updates:
- 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

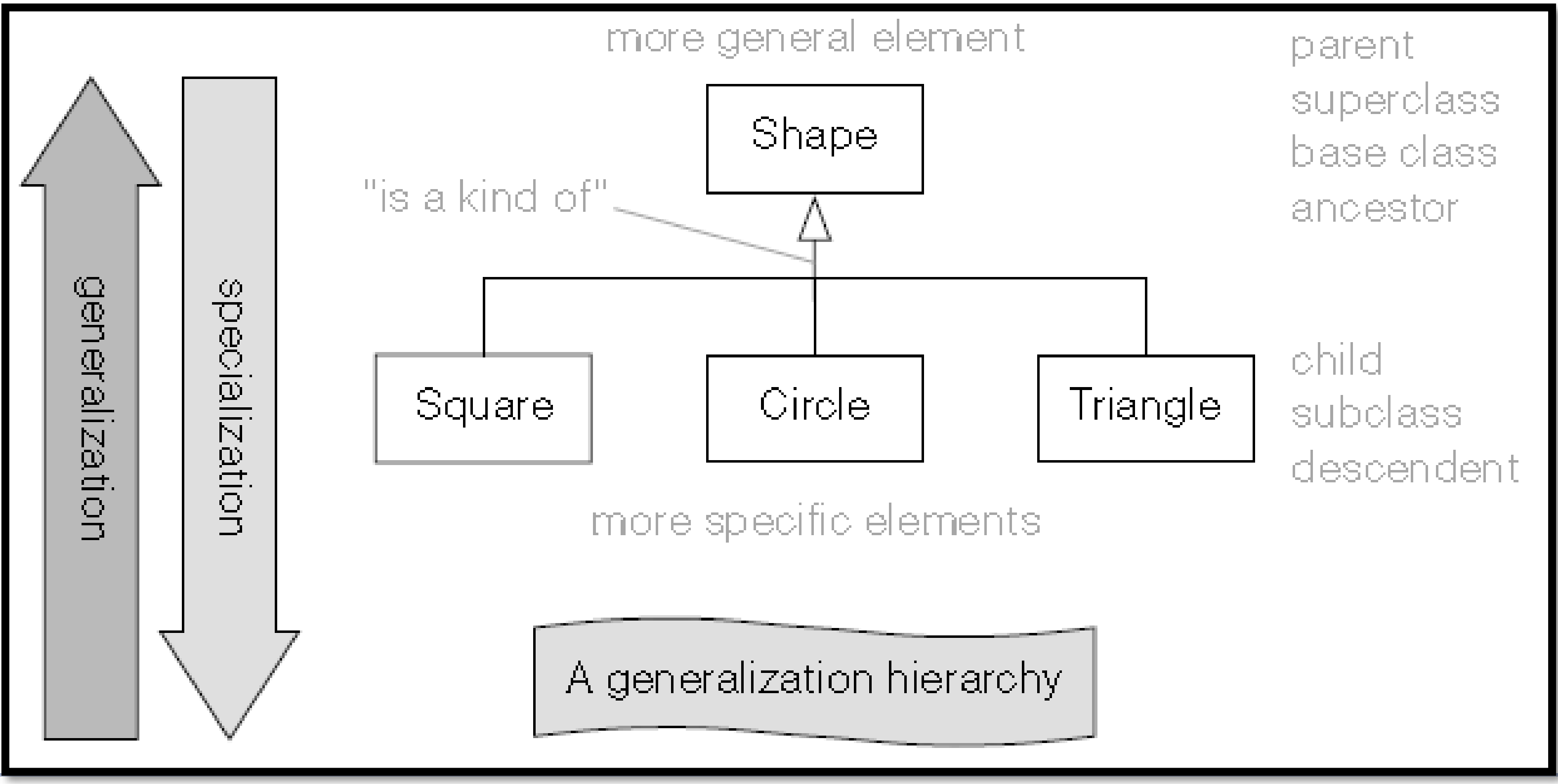
Reification of Behaviour

- Behavior written in code is rigid
- You can execute it, but cannot manipulate it at run time
- If you want to manipulate it like pass, store or modify the behavior at runtime you should reify it
- Reification is the promotion of something that is not an object into an object
- It dramatically expand the flexibility of a system

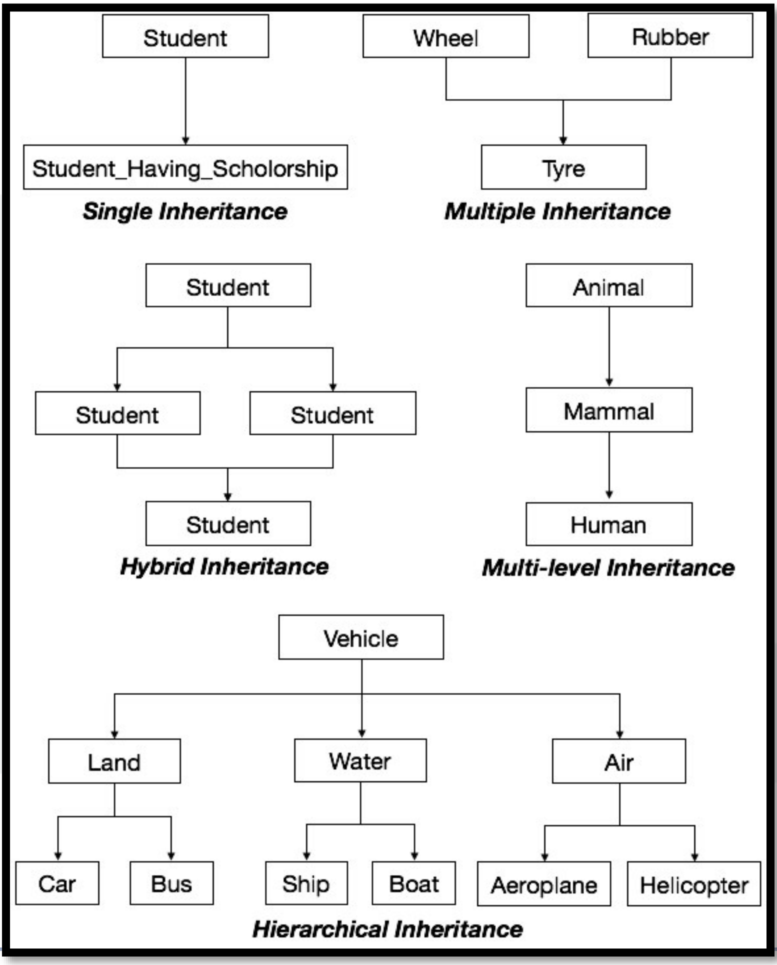
Adjustment of Inheritance

- Through the following steps: (refer the book)
- 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.

Adjustment of Inheritance



Adjustment of Inheritance



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

Design pattern: introduction and classification

- A design pattern provides a general reusable solution for the common problems occurs in software design. The patterns typically show relationships and interactions between classes or objects.
- The idea is to speed up the development process by providing well tested, proven development/design paradigm
- It’s not mandatory to implement design patterns in your project always. Design patterns are not meant for project development. Design patterns are meant for common problem-solving.

Goal

- Understand the purpose and usage of each design patterns. So, you will be able to pick and implement the correct pattern as needed.

Types of Design Patterns

- There are mainly three types of design patterns:

1. Creational
2. Structural
3. Behavioral

Creational

- These design patterns are all about class instantiation or object creation. These patterns can be further categorized into Class-creational patterns and object-creational patterns.
- While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.
- Creational design patterns are the Factory Method, Abstract Factory, Builder, Singleton, Object Pool, and Prototype.
- Use case of creational design pattern-
  1) Suppose a developer wants to create a simple DBConnection class to connect to a database and wants to access the database at multiple locations from code, generally what developer will do is create an instance of DBConnection class and use it for doing database operations wherever required.

Structural

- These design patterns are about organizing different classes and objects to form larger structures and provide new functionality.Structural design patterns are Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, and Proxy.
- Use Case Of Structural Design Pattern-
- 1) When 2 interfaces are not compatible with each other and want to make establish a relationship between them through an adapter its called adapter design pattern. Adapter pattern converts the interface of a class into another interface or classes the client expects that is adapter lets classes works together that could not otherwise because of incompatibility. so in these type of incompatible scenarios, we can go for the adapter pattern.

Behavioral

- Behavioral patterns are about identifying common communication patterns between objects and realize these patterns.Behavioral patterns are Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor
- Use Case of Behavioral Design Pattern-
- Template pattern defines the skeleton of an algorithm in an operation deferring some steps to sub-classes, Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure. say for an example in your project you want the behavior of the module can be extended, such that we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications. However, No one is allowed to make source code changes to it. it means you can add but can’t modify the structure in those scenarios a developer can approach template design pattern.

Parul® University

DIGITAL LEARNING CONTENT

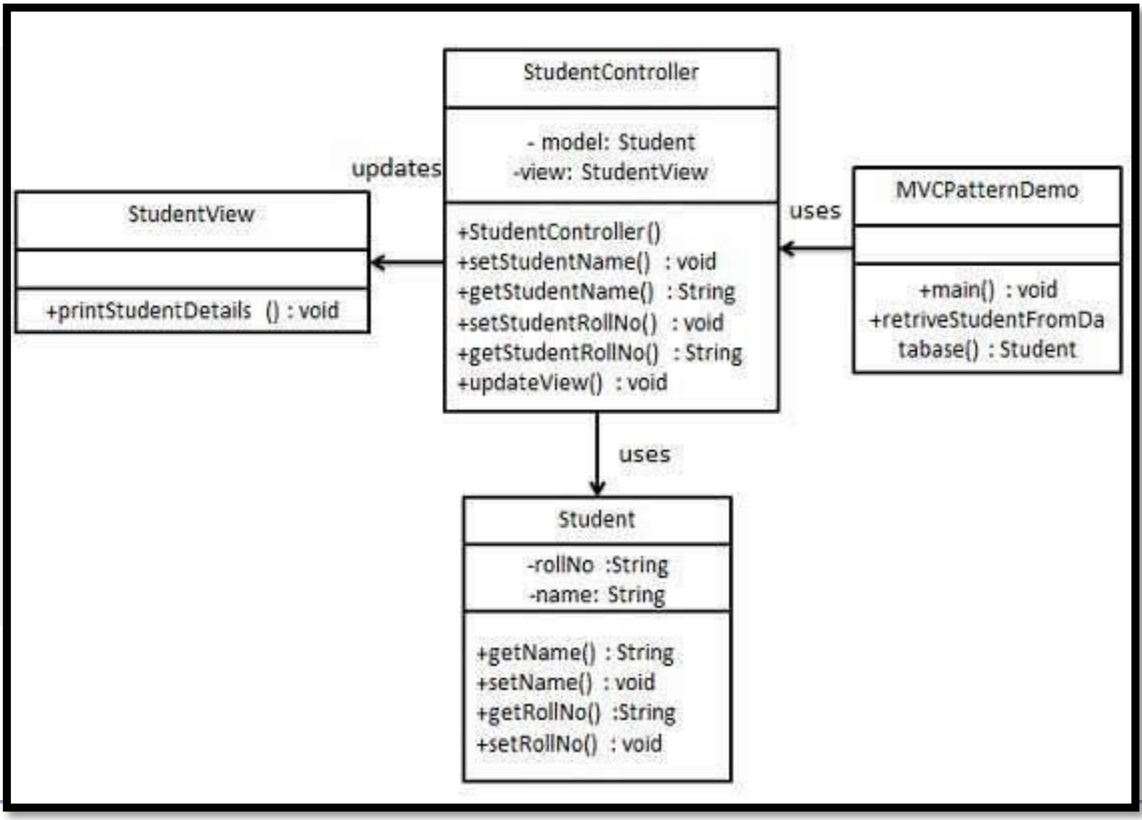
Case Study of Model View Controller (MVC).

- MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.
- **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
- **View** - View represents the visualization of the data that model contains.
- **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

Implementation

- We are going to create a *Student* object acting as a model.*StudentView* will be a view class which can print student details on console and *StudentController* is the controller class responsible to store data in *Student* object and update view *StudentView* accordingly.
- *MVCPatternDemo*, our demo class, will use *StudentController* to demonstrate use of MVC pattern.

MVC


StudentController
- model: Student
-view: StudentView
+StudentController()
+setStudentName() : void
+getStudentName() : String
+setStudentRollNo() : void
+getStudentRollNo() : String
+updateView() : void
updates
StudentView
+printStudentDetails () : void
uses
MVCPatternDemo
+main() : void
+retriveStudentFromDatabase() : Student
uses
Student
-rollNo :String
-name: String
+getName() : String
+setName() : void
+getRollNo() :String
+setRollNo() : void

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in