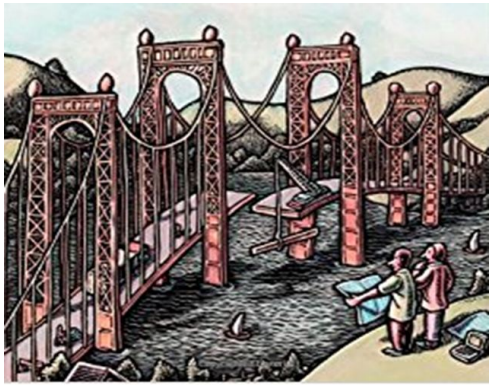
Object Oriented Analysis and Design using Java

Dr H L Phalachandra and Prof. Spurthi N Anjan
Department of Computer Science and Engineering

Object Oriented Analysis and Design using Java

GRASP: Object Oriented Design Patterns



Prof. Spurthi N Anjan
Department of Computer Science and Engineering

Object Oriented Analysis and Design using Java

Unit-03 : Dynamic Models, Diagrams and Architecture design and principles

PES UNIVERSITY ONLINE

Class-32 GRASP and its application to Object Design, Creator

Design Principles

- If we need to build object oriented systems which are maintainable, extensible and modular, it would be best done using well known principles and practices as part of the OO design.
- These address problems or potential issues which developers will face and provide some principles or approaches to address the same or patterns which can be used to get over the same.
- These techniques have not been invented to create new ways of working, but to better document and standardize old, tried-and-tested programming principles in object-oriented design.

Design Principles

- GRASP is an acronym created by Craig Larman to encompass nine object-oriented design principles related to creating responsibilities for classes.
- GRASP is an attempt to document what expert designers probably know intuitively.
- **GRASP**

  General Responsibility Assignment Software Patterns (or Principles)
- A collection of general objected-oriented design patterns related to assigning defining objects
- Originally described as a collection by Craig Larman in Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design, 1st edition, in 1997.

Why GRASP?

- Traditionally in Object-Oriented Programming (OOP), object design is glossed over –

  E.g., think of nouns and convert to objects; think of verbs and convert to methods –

  Or even: After requirements analysis and creation of a domain model, just create objects and their methods to fulfill requirements

  (Oh. ...Ok. Poor inexperienced OO developers.)

- UML is just a language—it expresses an OO design but for the most part does not provide guidance
- Per Larman, GRASP helps one "understand essential object design and apply reasoning in a methodical, rational, explainable way."

What is Responsibility

is a contract / obligation that a class / module / component must accomplish

- Responsibility can be:

  accomplished by a single object

  or a group of object collaboratively accomplish a responsibility.

- GRASP helps us in deciding which responsibility should be assigned to which object/class.
- Identify the objects and responsibilities from the problem domain, and also identify how objects interact with each other.
- Define blue print for those objects – i.e. class with methods implementing those responsibilities.

Responsibility

Design holds the responsibility of knowing the data & associated data and the actions or behaviours associated with data

- Knowing in terms of its
  - Private state
  - Computed state
- Expected Behaviour in terms
- Send messages by itself and modifying its private state
  - Instantiate another object
  - Send messages to another object

Responsibility knowing and Doing

- This involves design supporting the “doing” responsibilities
  - Doing something itself, such as creation an object or doing a calculation.
  - Initiating action in other objects
  - Controlling and coordinating activities in other objects.
- This involves the design supporting the “knowing” responsibilities
  - Knowing about encapsulated data
  - Knowing about related objects.
  - Knowing about things it can derive or calculate

PES UNIVERSITY ONLINE

GRASP Principles (9 of them)

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. Indirection
7. Polymorphism
8. Protected Variations
9. Pure Fabrication

1. Creator Principles

- Who creates an Object? Or who should create a new instance of some class?
- "Container" object creates "contained" objects.
- Decide who can be creator based on the objects association and their interaction.

- Problem: Who should be responsible for creating a new instance of some class?

1. Creator Principles

Solution:

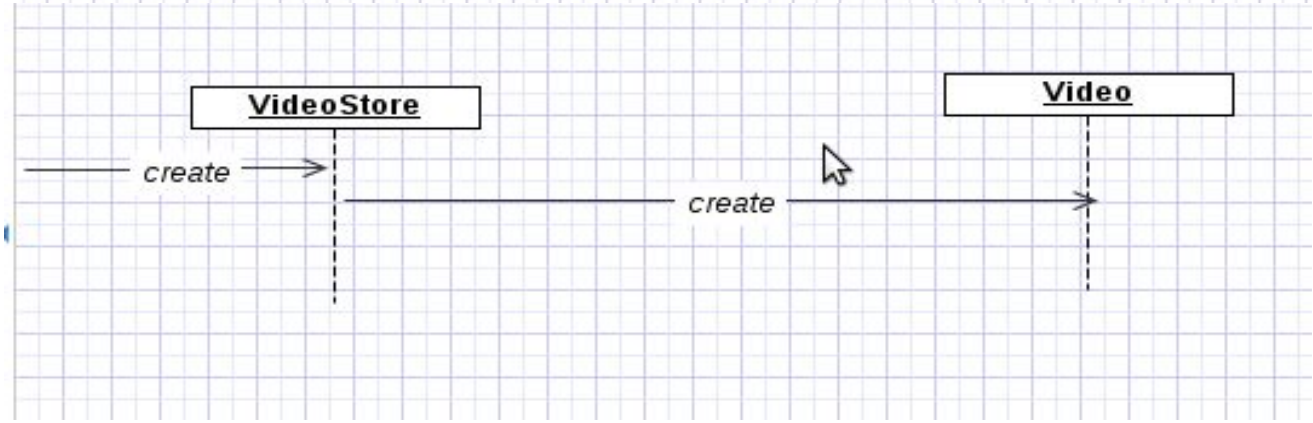
- Assign class B the responsibility to create an instance of class A if one of the below is true (the more the better).
- If more than one option applies, usually prefer a class B which aggregates or contains A.
  - B contains or is composed of A.
  - B records A.
  - B closely uses A.
  - B has the initializing data for A that will be passed to A when it is created.
- Thus B is an Expert with respect to creating A.

1. Creator Example

- Consider VideoStore and Video in that store.
- VideoStore has an aggregation association with Video. i.e VideoStore is the container and the Video is the contained object.
- So, we can instantiate video object in a VideoStore class

Class Relation

Sequence



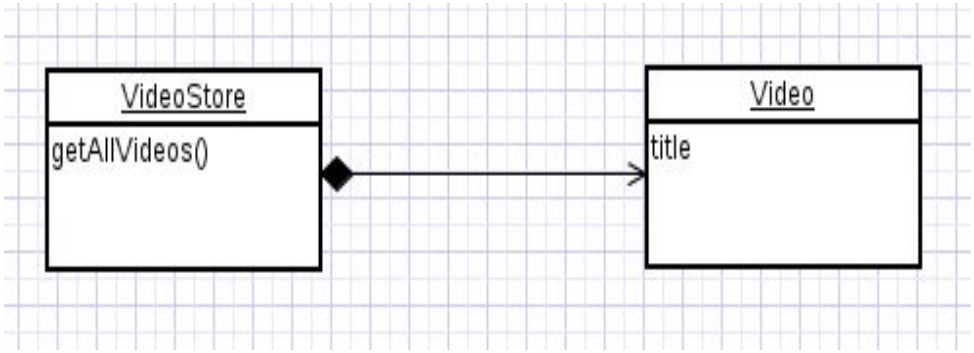
2. Information Expert

- Given an object O, which of the responsibilities can be assigned to O?
- Expert principle says – assign those responsibilities to a class which has the information to fulfill that responsibility.
- They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.

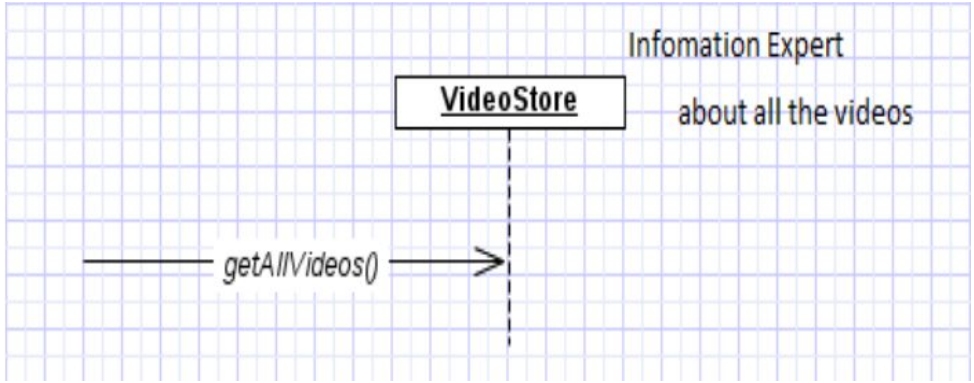
2. Information Expert

- Assume we need to get all the videos of a VideoStore.
- Since VideoStore knows about all the videos (catalog), we can assign this responsibility of giving all the videos can be assigned to VideoStore class.
- VideoStore is the information expert.

Class Relation



Sequence

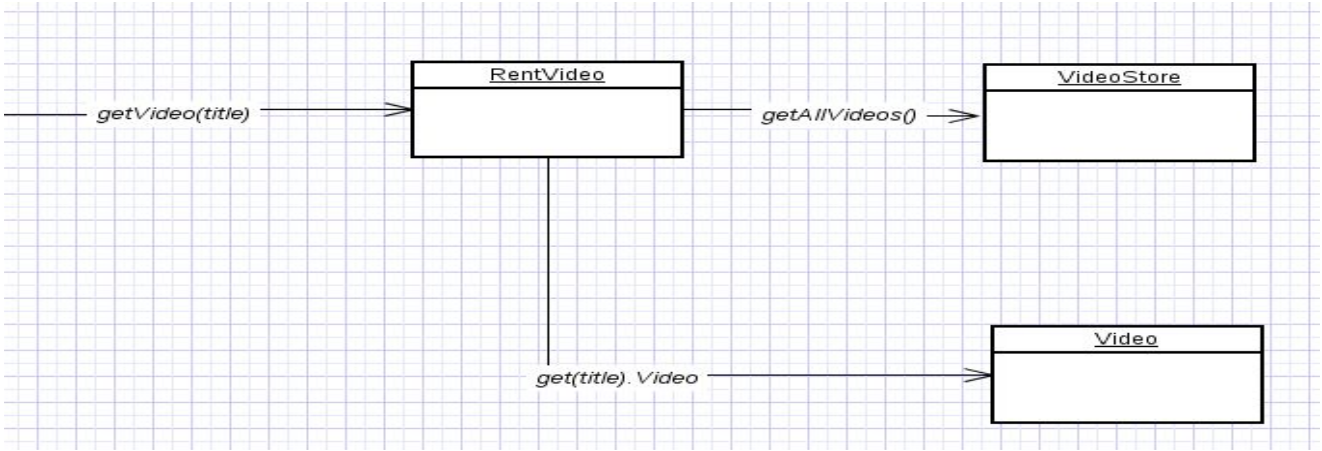


3. Low Coupling

- How strongly the objects are connected to each other? Coupling – object depending on other object.
- When depended upon element changes, it affects the dependant also
- Low Coupling – How can we reduce the impact of change in depended upon elements on dependant elements.
- Prefer low coupling – assign responsibilities so that coupling remain low.
- Minimizes the dependency hence making system maintainable, efficient and code reusable
- Two elements are coupled, if – One element has aggregation/composition association with another element. – One element implements/extends other element.

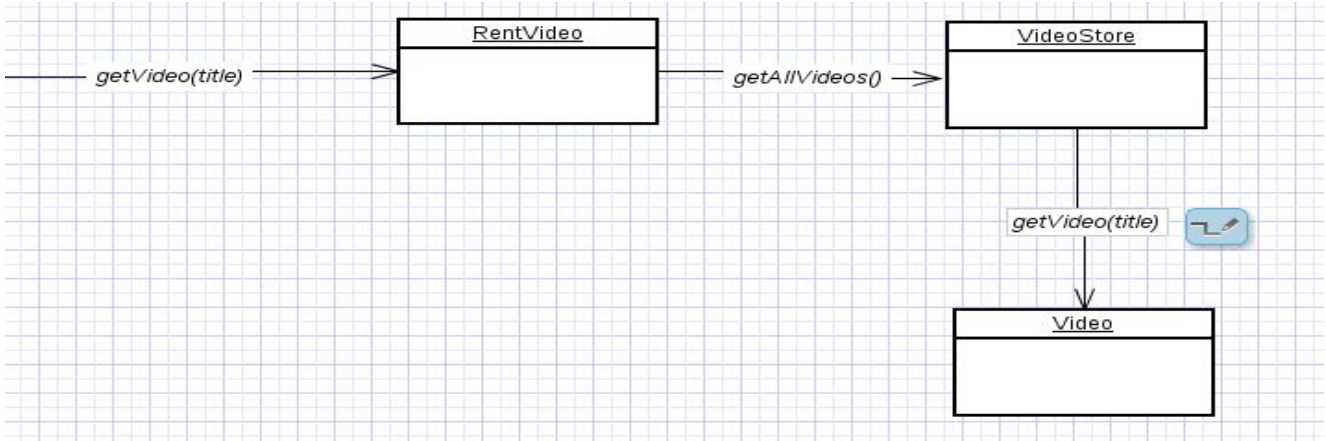
3. Low Coupling Example

High Coupling



here class RentVideo knows about both VideoStore and Video objects. Rent is depending on both the classes.

Low Coupling

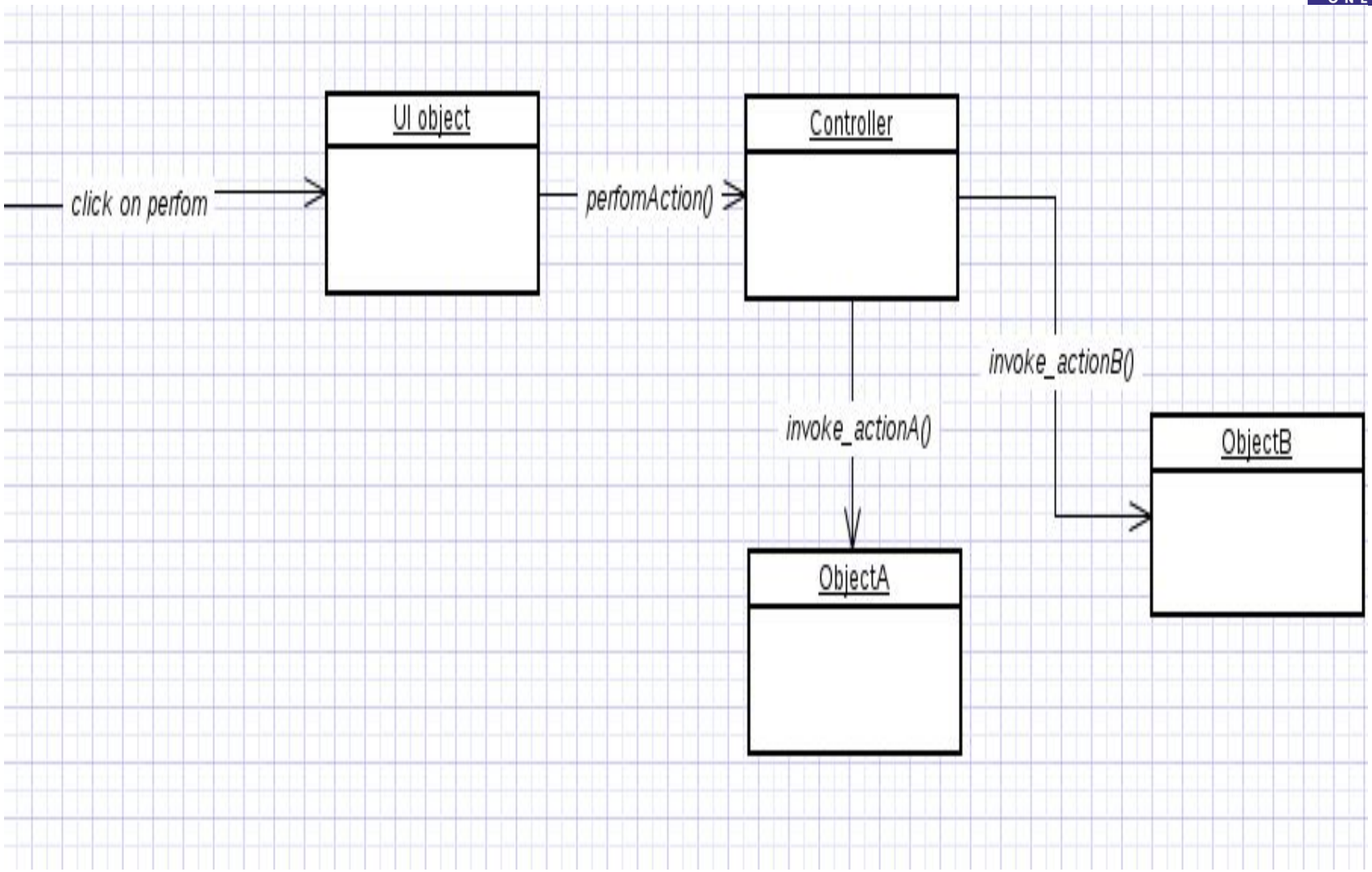


VideoStore and Video class are coupled, and Rent is coupled with VideoStore. **Thus providing low coupling.**

4. Controller

- Deals with how to delegate the request from the UI layer objects to domain layer objects .
- Controller helps in minimizing the dependency between GUI components and the system operation classes which represent a set of system level operation
- When a request comes from UI layer object, Controller as a pattern helps us in determining what is that first object that should receive the message from the UI layer objects.
- This object is called controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request.
- It delegates the work to other class and coordinates the overall activity.
- Benefits
  - can reuse this controller class.
  - Can use to maintain the state of the use case.

4. Controller Examples



5. High Cohesion

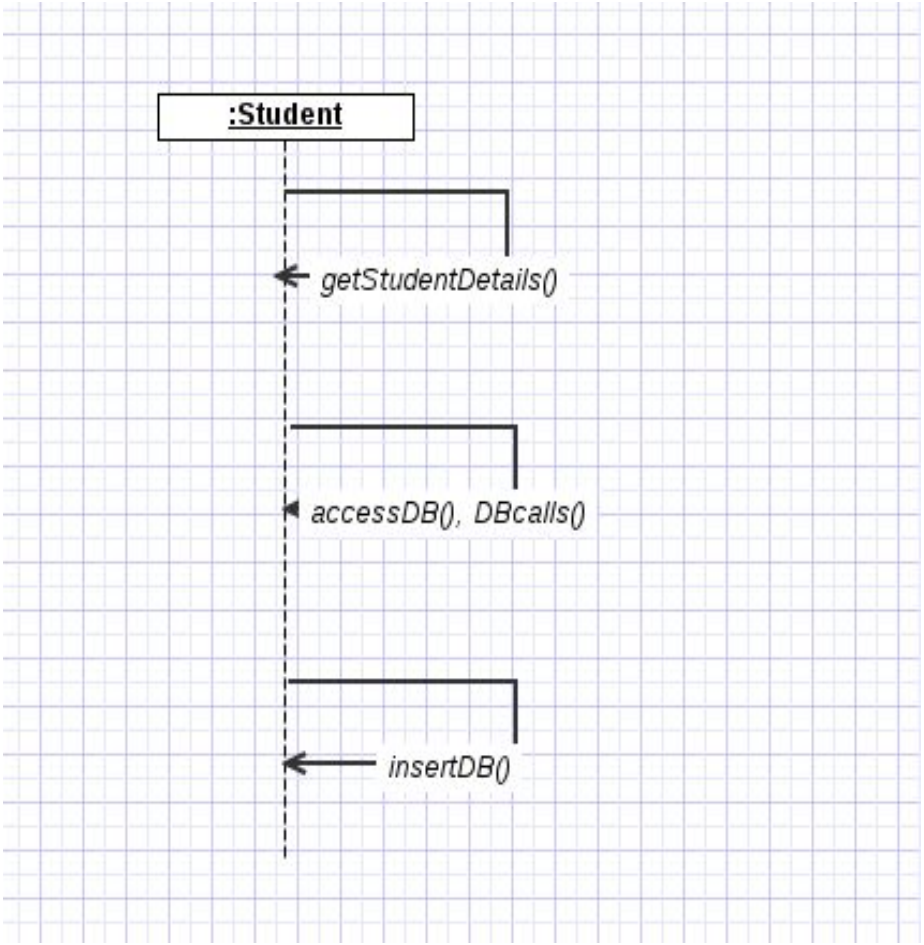
- How are the operations of any element are functionally related?
- Related responsibilities into one manageable unit.
- Prefer high cohesion
- Clearly defines the purpose of the element

Benefits:

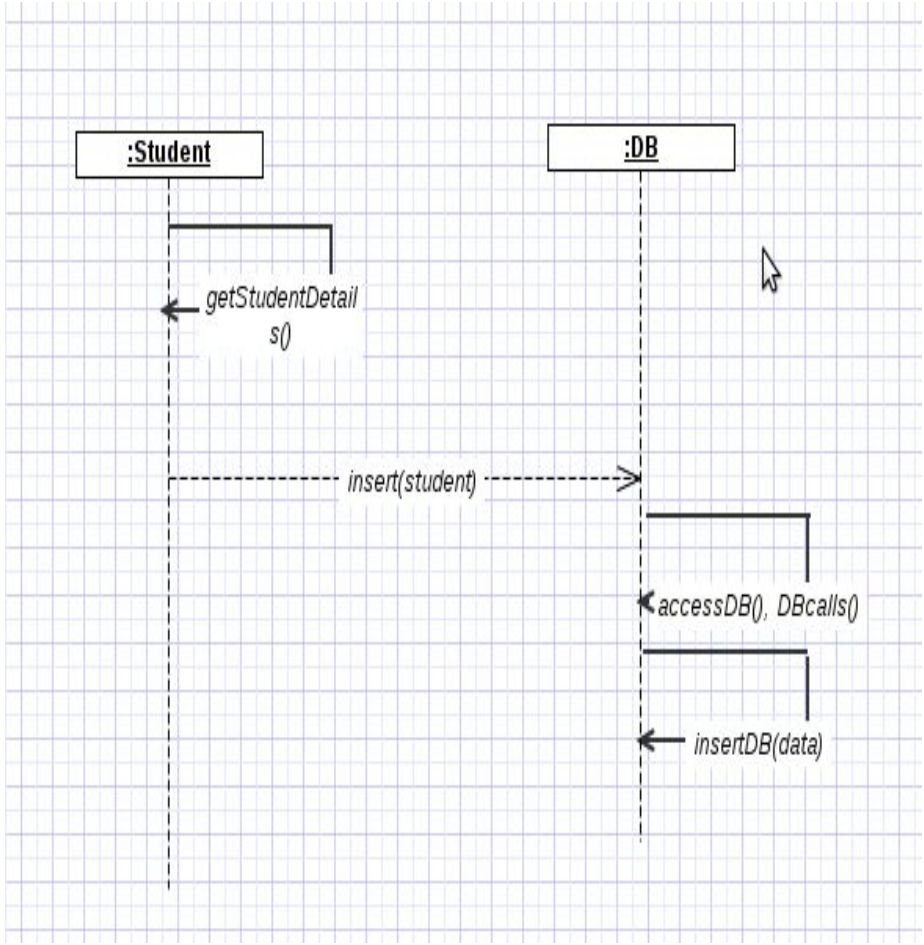
- Easily understandable and maintainable.
- Code reuse
- Low coupling

5. High Cohesion Example

Low Cohesion



Higher Cohesion



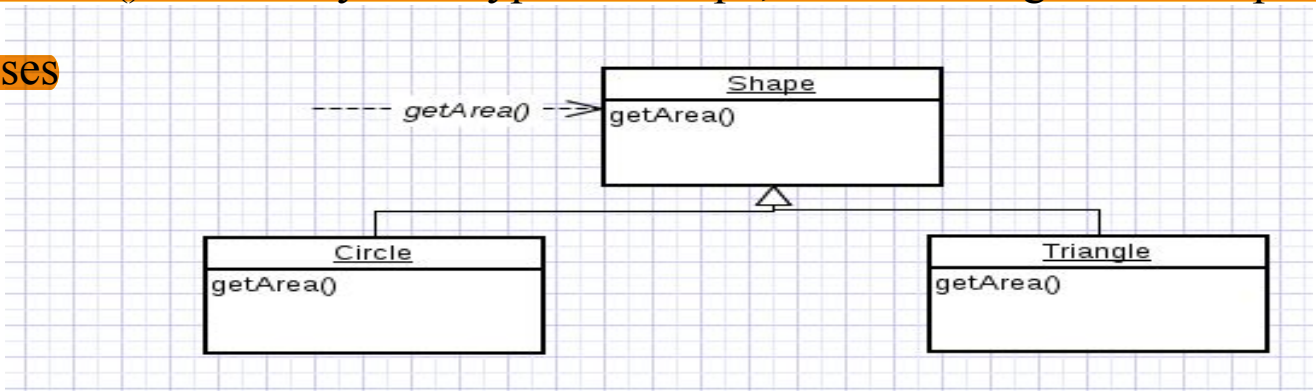
6. Polymorphism

- How to handle related but varying elements based on element type?
- Polymorphism guides us in deciding which object is responsible for handling those varying elements.

Benefits: Handling new variations will become easy.

Example:

- The getArea() varies by the type of shape, so we assign that responsibility to the subclasses



- By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle

7. Pure Fabrication

- Use of a pure fabrication class that does not represent a concept in the problem domain, to make up a assigned set of related responsibilities.
- Provides a highly cohesive set of activities.
- Behavioral decomposed – implements some algorithm.

Examples: Adapter, Strategy design patterns

- **Benefits:** High cohesion, low coupling and can reuse this class

Eg.

- Suppose we have a class for Shape and if we must store the shape data in a database.
- If we put this responsibility in Shape class, there will be many database related operations thus making Shape incohesive.
- So, create a fabricated class DBStore which is responsible to perform all database operations.

Eg. logInterface which is responsible for logging information is also a good example

8. Indirection

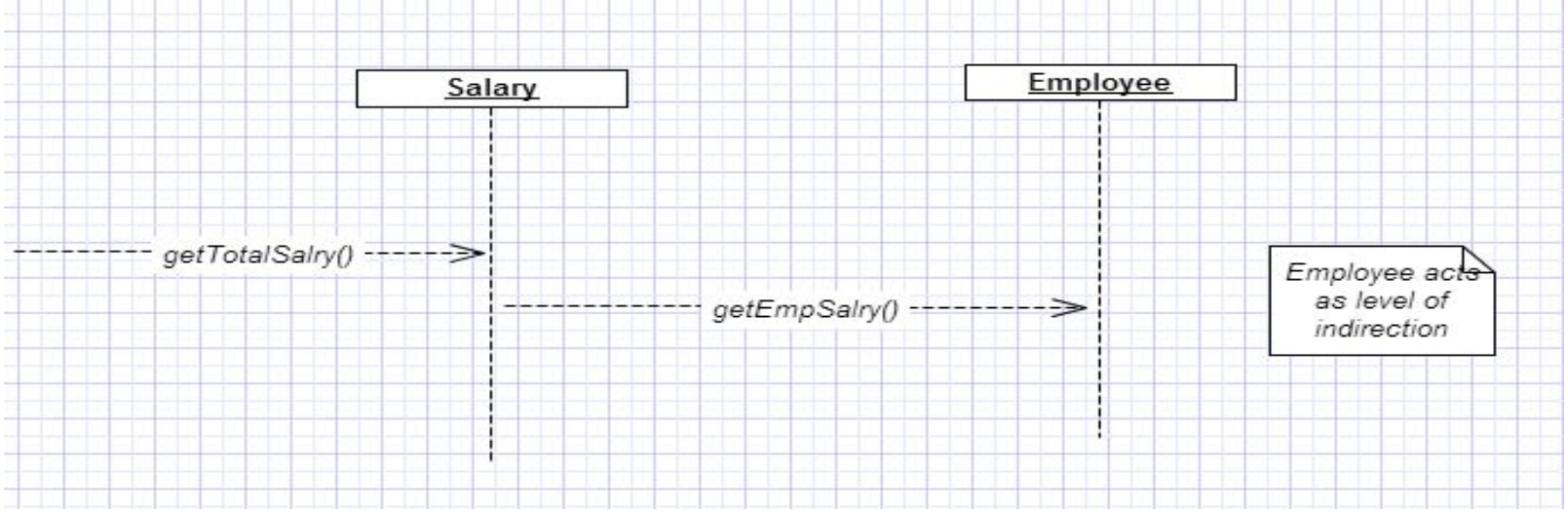
- How can we avoid a direct coupling between two or more elements.
- Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.

Benefits: Low coupling

Design Patterns : Adapter, Façade, Observer

Example : Here polymorphism illustrates indirection

- Class Employee provides a level of indirection to other units of the system.



9. Protected Variation

- How to avoid impact of variations of some elements on the other elements.
- Achieved by providing a well defined interface so that the there will be no affect on other units and then using polymorphism to create various implementations of this surface
- Provides flexibility and protection from variations.
- Provides more structured design.

Example: Interfaces-polymorphism

Usage

Problem: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
Solution: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

THANK YOU

Prof Spurthi N Anjan
Department of Computer Science and Engineering

spurthianjan@pes.edu