

GRASP Design Principles

By Danya Rao

GRASP

- stands for General Responsibility Assignment Software Patterns
- guides in assigning responsibilities to collaborating objects.
- 9 GRASP patterns

Creator

Information Expert

Low Coupling

Controller

High Cohesion

Indirection

Polymorphism

Protected Variations

Pure Fabrication

Responsibility:

- 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.

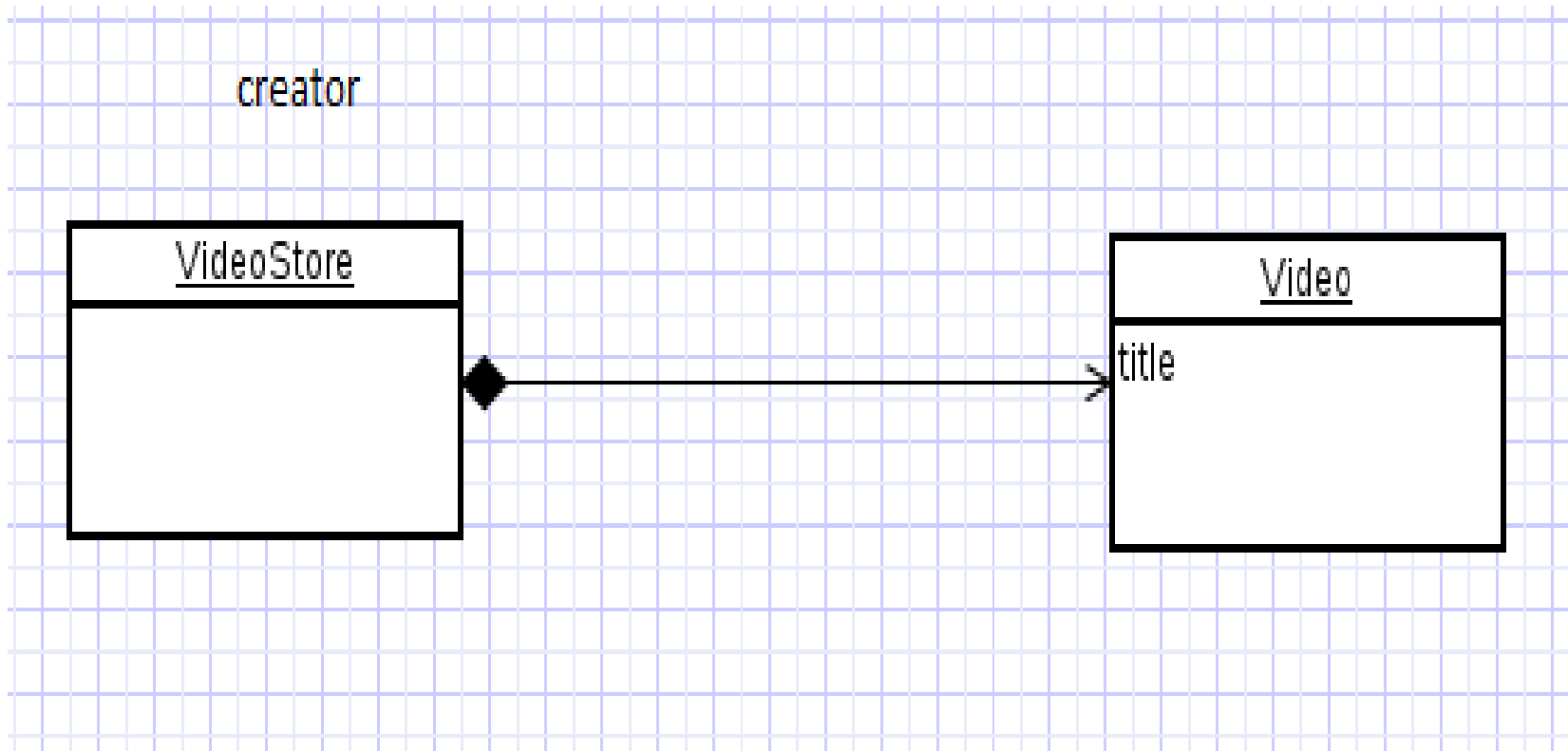
Creator

- 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.

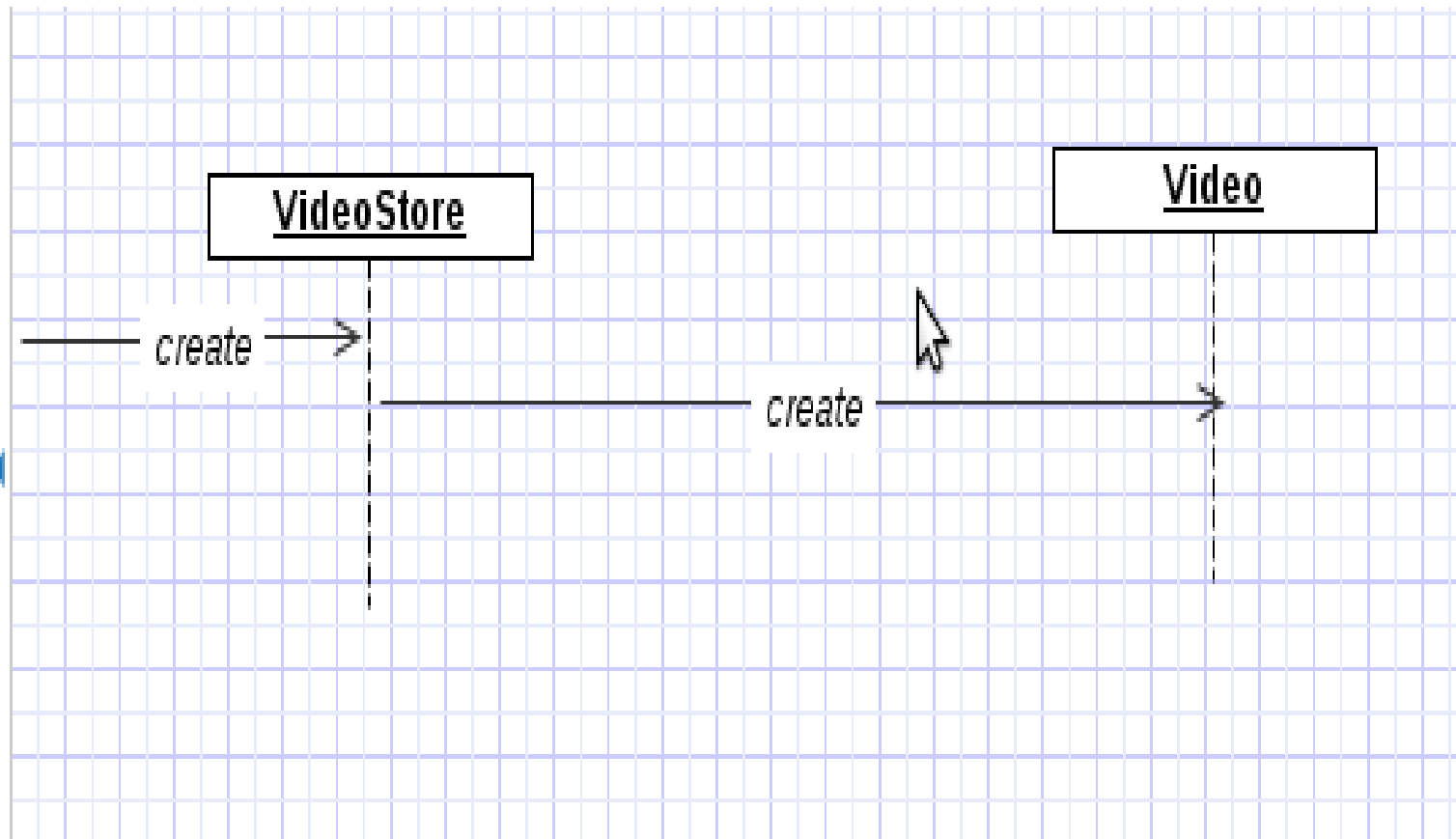
Example for Creator

- 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 VideoStore class

Example diagram



Example for creator



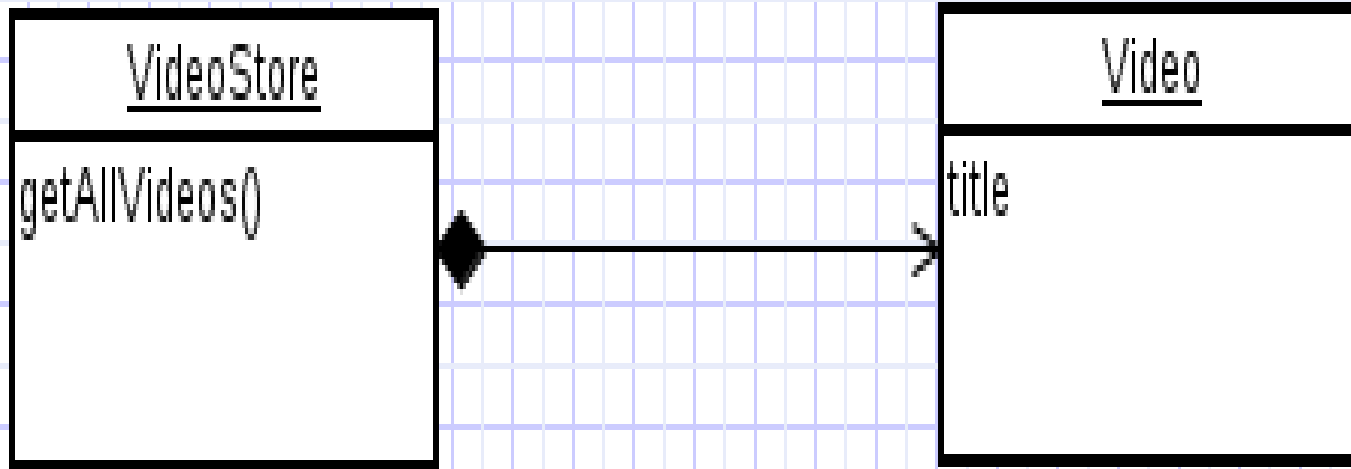
Expert

- Given an object o , which responsibilities can be assigned to o ?
- Expert principle says – assign those responsibilities to o for which o 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.

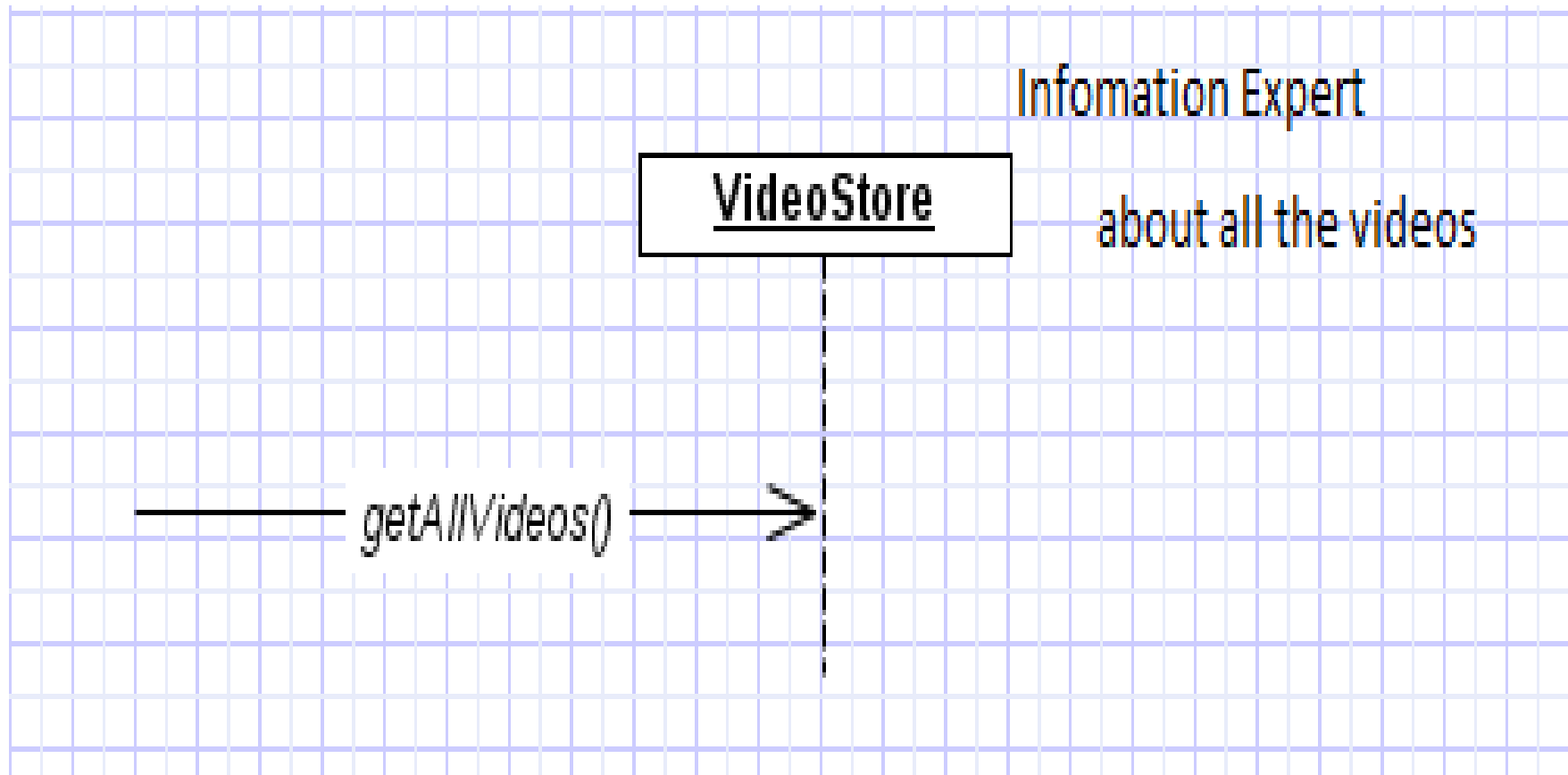
Example for Expert

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

Example for Expert



Example for Expert



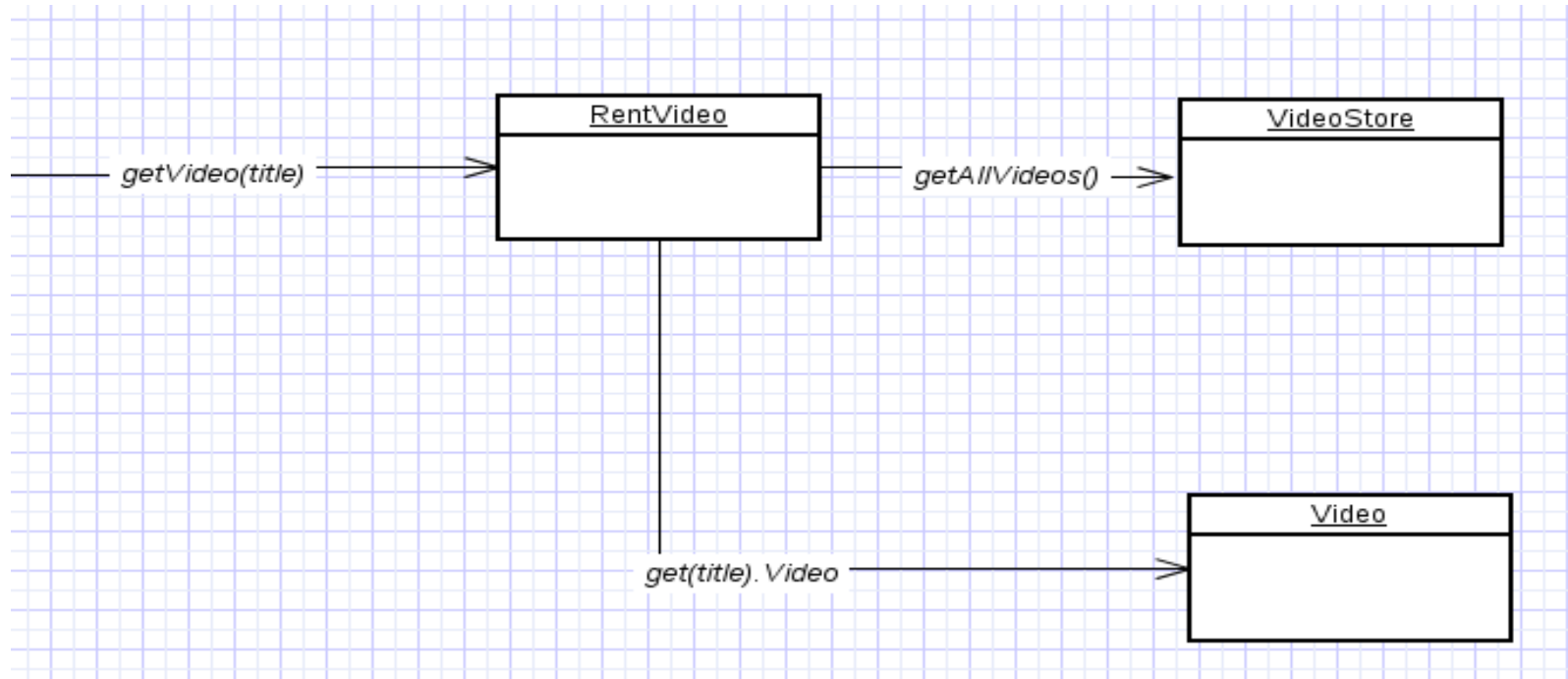
Low Coupling

- 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

Low coupling

- Two elements are coupled, if
 - One element has aggregation/composition association with another element.
 - One element implements/extends other element.

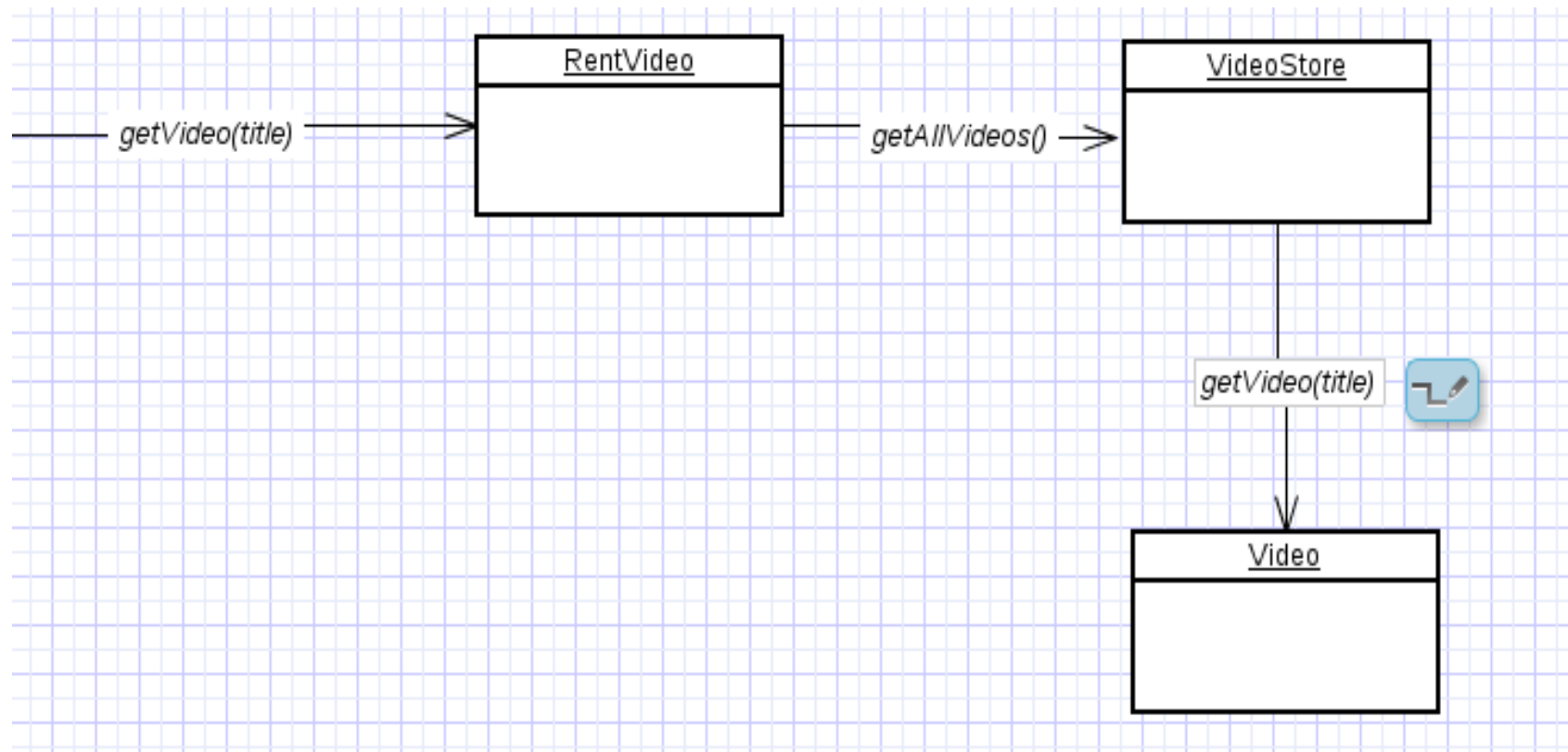
Example for poor coupling



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

Example for low coupling

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



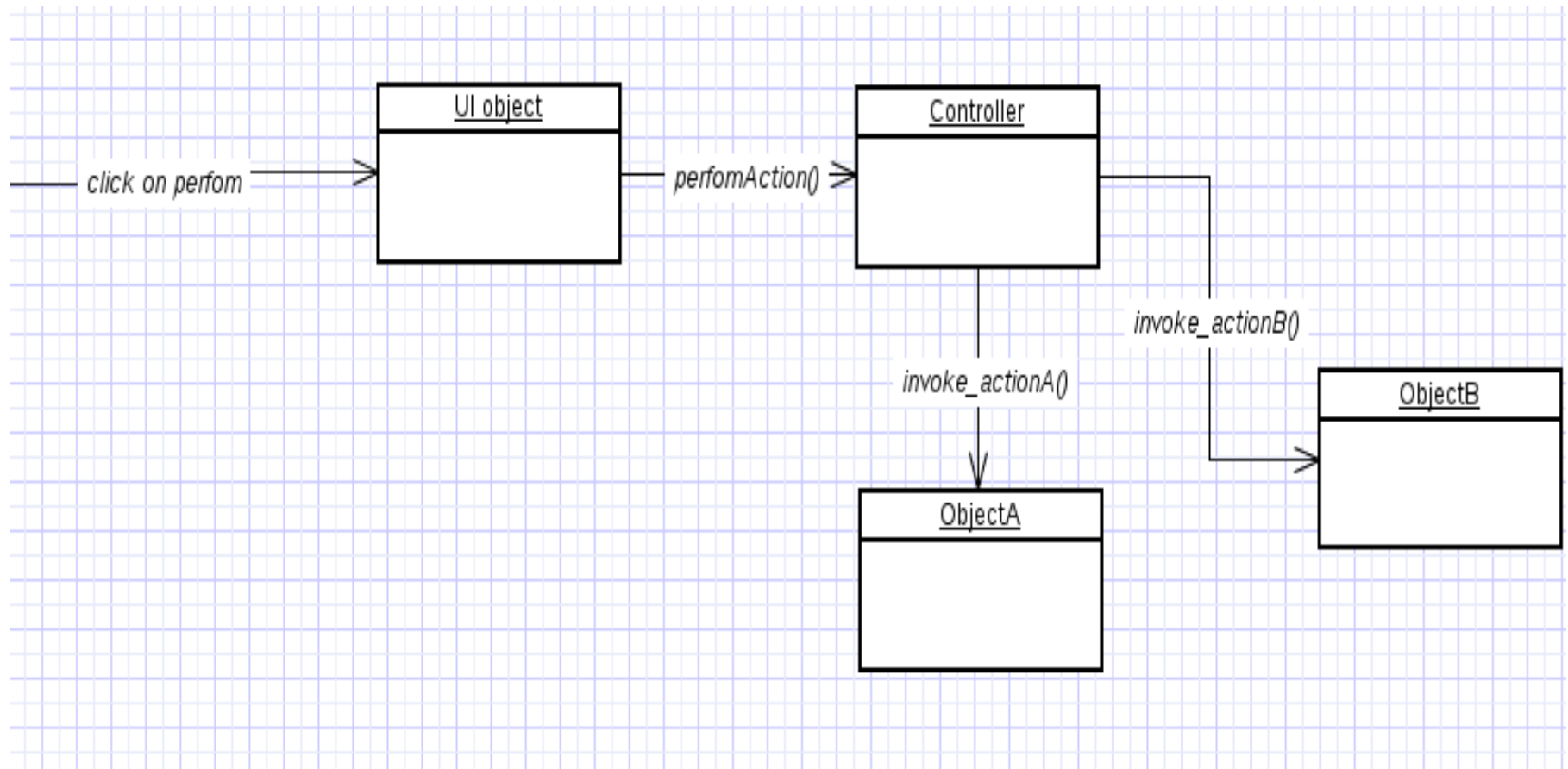
Controller

- Deals with how to delegate the request from the UI layer objects to domain layer objects.
- when a request comes from UI layer object, Controller pattern helps us in determining what is that first object that 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.

Controller

- We can make an object as Controller, if
 - Object represents the overall system (facade controller)
 - Object represent a use case, handling a sequence of operations (session controller).
- Benefits
 - can reuse this controller class.
 - Can use to maintain the state of the use case.
 - Can control the sequence of the activities

Example for Controller



Bloated Controllers

- Controller class is called bloated, if
 - The class is overloaded with too many responsibilities.

Solution – Add more controllers

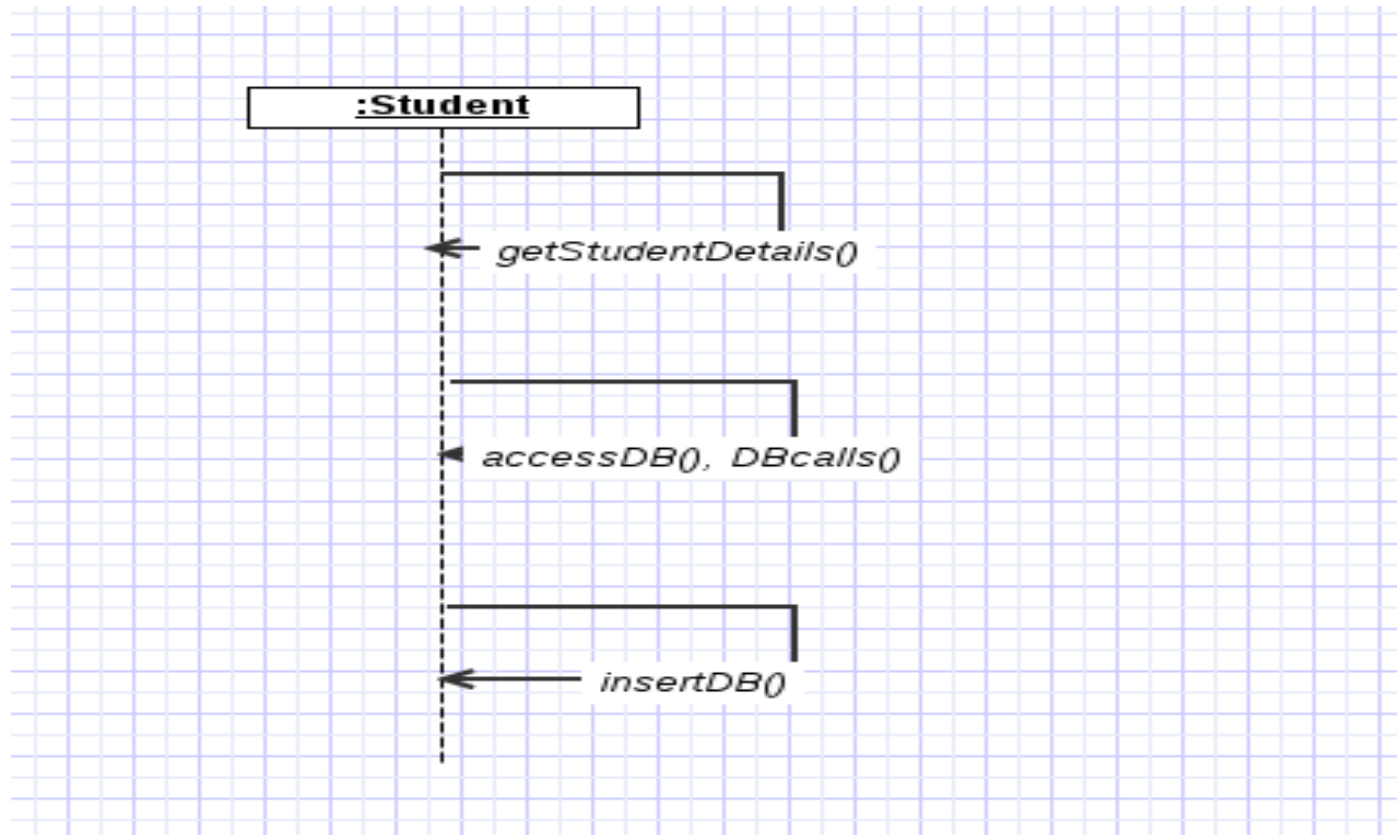
- Controller class also performing many tasks instead of delegating to other class.

Solution – controller class has to delegate things to others.

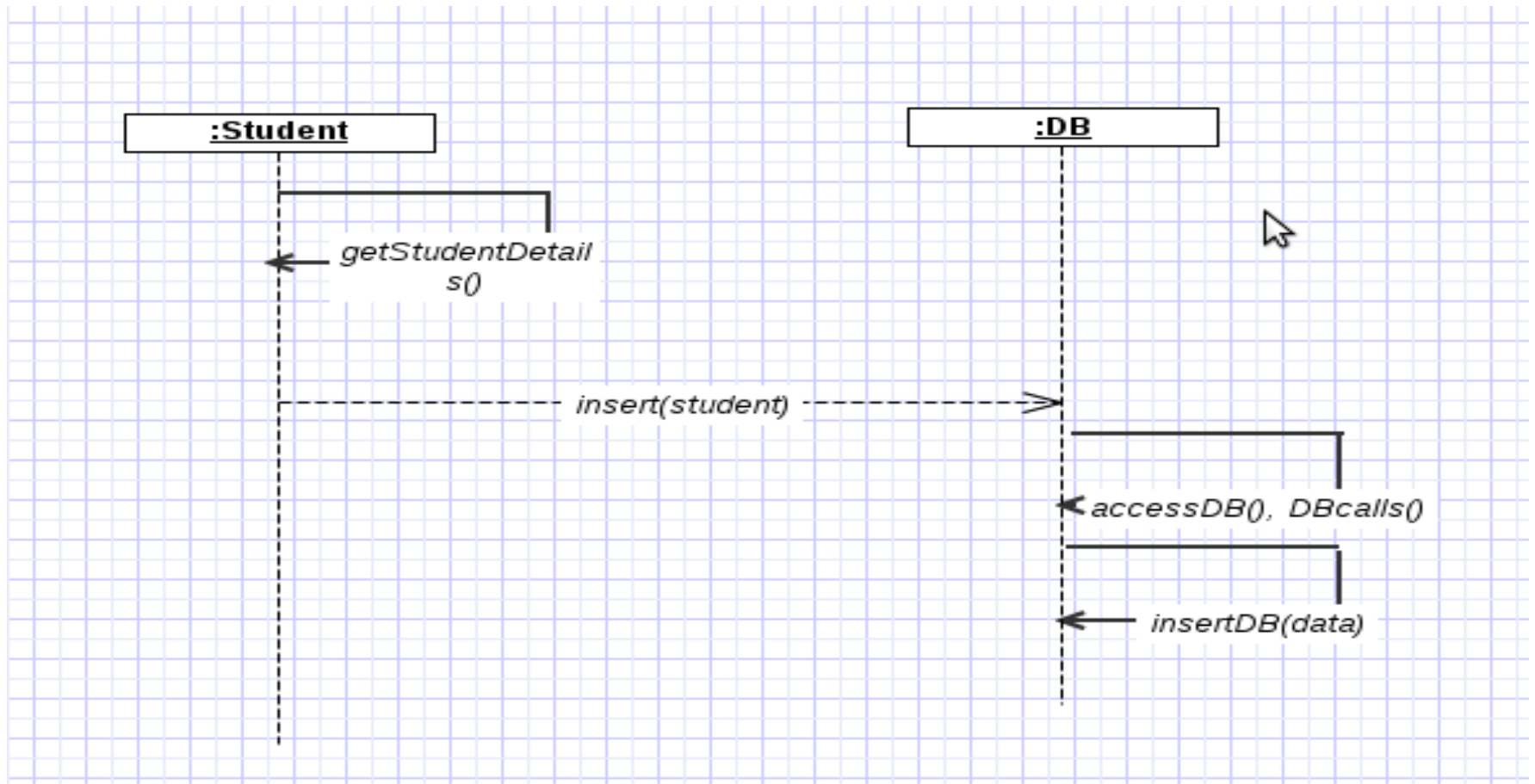
High Cohesion

- How are the operations of any element are functionally related?
- Related responsibilities in to one manageable unit.
- Prefer high cohesion
- Clearly defines the purpose of the element
- Benefits
 - Easily understandable and maintainable.
 - Code reuse
 - Low coupling

Example for low cohesion



Example for High Cohesion

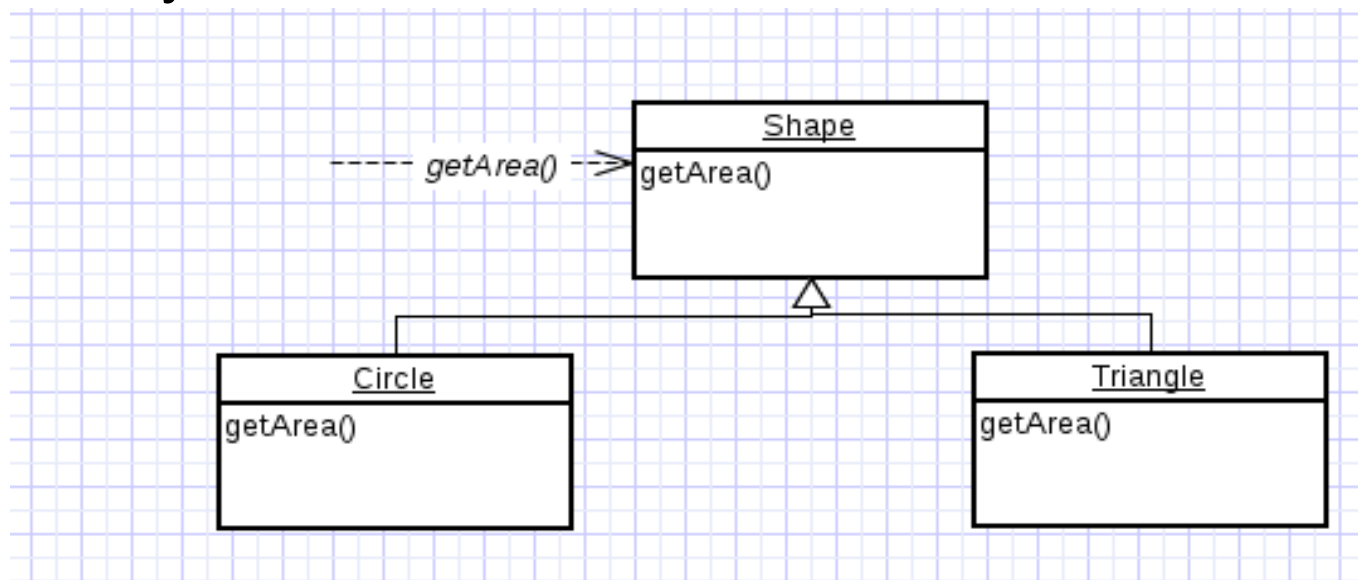


Polymorphism

- 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 for Polymorphism

- 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`.

Pure Fabrication

- Fabricated class/ artificial class – assign set of related responsibilities that doesn't represent any domain object.
- Provides a highly cohesive set of activities.
- Behavioral decomposed – implements some algorithm.
- Examples: Adapter, Strategy
- Benefits: High cohesion, low coupling and can reuse this class.

Example

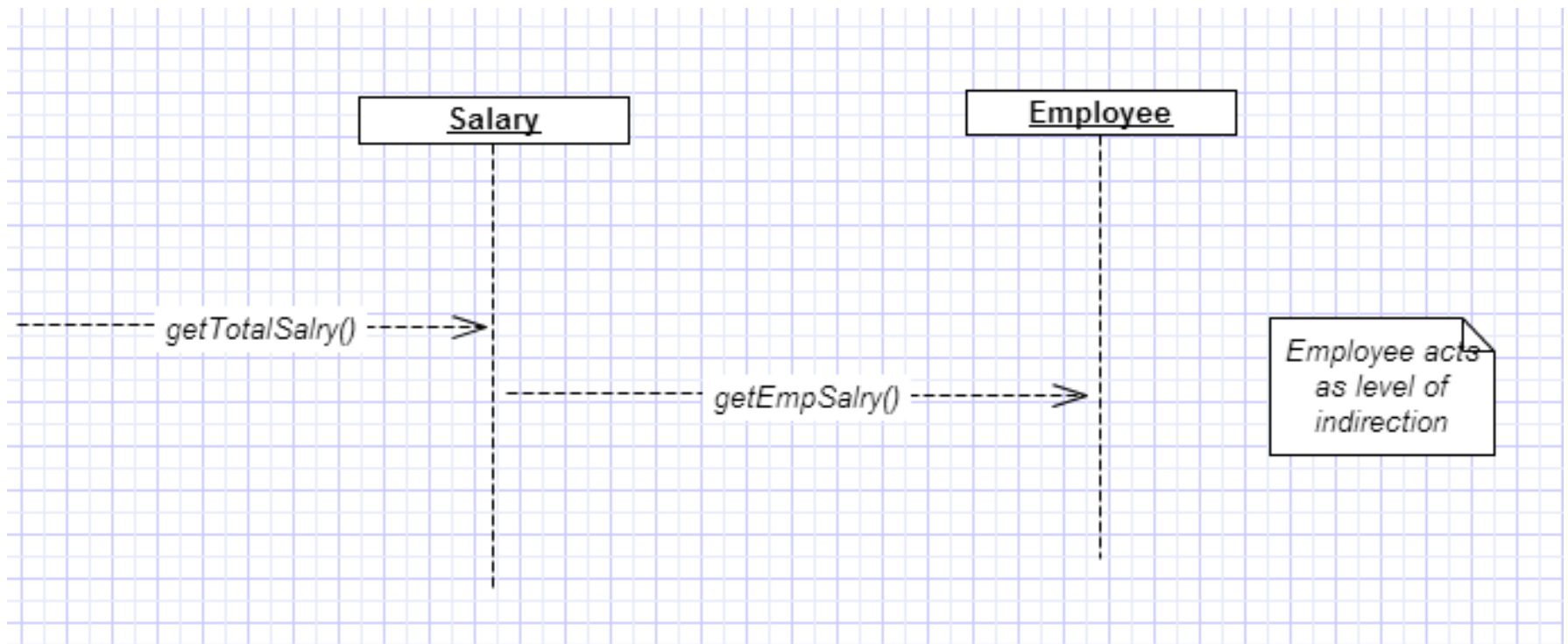
- Suppose we Shape class, if we must store the shape data in a database.
- If we put this responsibility in Shape class, there will be many database related operations thus making Shape incohesive.
- So, create a fabricated class DBStore which is responsible to perform all database operations.
- Similarly logInterface which is responsible for logging information is also a good example for Pure Fabrication.

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
- Example: Adapter, Facade, Observer

Example for Indirection

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



Protected Variation

- How to avoid impact of variations of some elements on the other elements.
- It provides a well defined interface so that there will be no affect on other units.
- Provides flexibility and protection from variations.
- Provides more structured design.
- Example: polymorphism, data encapsulation, interfaces

Reference

- Applying UML and Patterns, Third Edition,
Craig Larman