OBJECT ORIENTED ANALYSIS & DESIGN
DATA STRUCTURES & ALGORITHMS

Best Practices

LearnOA®

The Beginning of Patterns

- Christopher Alexander, architect
  - ✓ A Pattern Language--Towns, Buildings, Construction
  - ✓ Timeless Way of Building (1979)
  - ✓ “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

- **Other patterns:** novels (tragic, romantic, crime), movies genres (drama, comedy, documentary)

SOLID Principles

- Acronym of acronyms:
  - ✓ SRP: Single Responsibility Principle
  - ✓ OCP: Open-Closed Principle
  - ✓ LSP: Liskov Substitution Principle
  - ✓ ISP: Interface Segregation Principle
  - ✓ DIP: Dependency Inversion Principle

- Basically, a set of principles for object-oriented design (with focus on designing the classes).

Benefits of SOLID

- Provides a principled way to manage dependency.
- Serves as a solid foundation for OOD upon which more complicated design patterns can be built upon and incorporated naturally.
- Results in code that are flexible, robust, and reusable.

Understanding SOLID

SRP: “A class should have one, and only one, reason to change”.

OCP: “You should be able to extend a class’s behavior, without modifying it”

LSP: “Derived classes must be substitutable for their base classes.”

ISP: “Make fine grained interfaces that are client specific.”

DIP: “Depend on abstrations, not on concretions.”

SRP: Single Responsibility Principle

- Example: Rectangle class with draw() and area()
- Computational geometry now depends on GUI, via Rectangle.
- Any changes to Rectangle due to Graphical application necessitates rebuild, retest, etc. of Comp. geometry app.
- Solution: Take the purely computational part of the Rectangle class and create a new class “Geometric Rectangle”.
- All changes regarding graphical display can then be localized into the Rectangle class.

SRP: another example

- Modem: dial(), hangup(), send(), recv(), ...
- However, there are two separate kinds of functions that can change for different reasons:
  - ✓ Connection-related
  - ✓ Data communication-related
- These two should be separated.
- Recall that “Responsibility” == “a reason to change”.

- “SRP is the simplest of the principles, and one of the hardest to get right.”
- We tend to join responsibilities together.
- SRP says we need to go against this tendency.

OCP: Open-Closed Principle

“All systems change during their life cycles.” (Ivar Jacobson).

- “Software entities should be open for extension, but closed for modification.” (variation on Bertrand Meyer’s idea).
- Goal: avoid a “cascade of changes to dependent modules”.
- When requirements change, you extend the behavior, not changing old code.

OCP: Data-Driven Approach

In many cases, complete closure (closure to modification) may not possible.

- Data-driven approach can be taken to minimize and localize changes to a small region of code that only contain data, not code.
- For example, there can be a table that contains a specific ordering based on the requirements, where the requirements are expected to change.

OCP: Open-Closed Principle

OCP leads to many heuristics and conventions.

- Make all member variables private.
- No global variables, EVER.
- Run time type identification (e.g., dynamic cast) is dangerous.

- OCP is “at the heart of OOD”.
- Simply using an OOP is not enough: Need dedication to apply abstraction.
- OCP can greatly enhance reusability and maintainability.

LSP: Liskov Substitution Principle

- “Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.” (original idea due to Barbara Liskov).
- Violation means the user class’s need to know ALL implementation details of the derived classes of the base class.
- Violation of LSP leads to the violation of OCP.

LSP: example

Rectangle Class <− Square Class

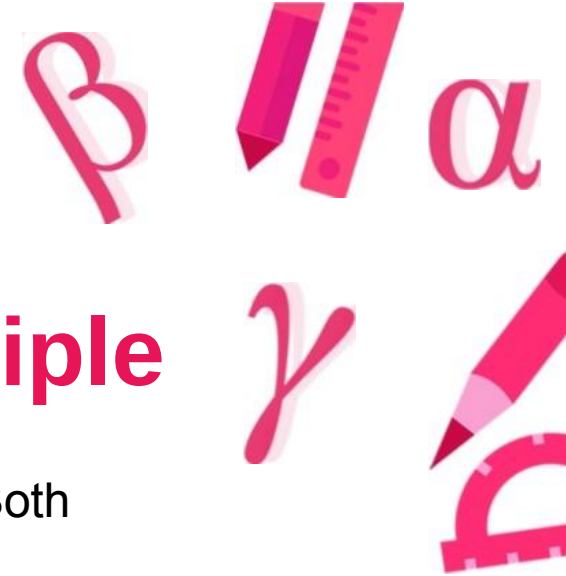
- Problem: setWidth(), setHeight() in Rectangle class are not a good fit for Square class.
- When Square class is used where Rectangle class is called for, behavior can be unpredictable, depending on implementation.
- Want either setWidth() or setHeight() to set both width and height in the Square class.
- LSP is violated when adding a derived class requires modifications of the base class.

LSP: summary

- Cannot assess vailidty of a class by just looking inside a class: We must see how it is used.
- “ISA relationship pertains to behavior”, extrinsic, public behavior!
  - ✓ Square is a Rectangle, but they behave differently, seen from the outside.
- For LSP to hold, ALL derived classes should conform to the behavior that the clients expect of the base classes.
- LSP is an important property that holds for all programs that conform to the Open-Closed principle.
- LSP encourages reuse of base types, and allows modifications in the derived class without damaging other components.

ISP: Interface Segregation Principle

- “Clients should not be forced to depend upon interfaces that they do not use.”
- Avoid “fat interfaces”.
- Fat interfaces: interfaces of a class that can be broken down into groups that server differnt set of clients.
- Clients depending on a subset of interfaces need to change when other clients using a different subset changes.



DIP: Dependency Inversion Principle

- “A. High level modules should not depend upon low level modules. Both should depend upon abstractions.”
- “ B. Abstractions should not depend upon details. Details should depend upon abstractions.”
- DIP is an out-growth of OCP and LSP.
- “Inversion”, because standard structured programming approaches make the higher level depend on lower level.

DIP: The Problem

- **Bad design:**
  - ✓ Hard to change (rigidity)
  - ✓ Unexpected parts break when changing code (fragility)
  - ✓ Hard to reuse (immobility)

- **Cause of bad design:**
  - ✓ Interdependence of the modules
  - ✓ Things can break in areas with NO conceptual relationship to the changed part.
  - ✓ Dependent on unnecessary detail.

DIP: Example

Copy(): uses ReadKeyboard() and WritePrinter(char c);

- Copy() is a general (high-level) functionality we want to reuse.
- The above design is tied to the specific set of hardware, so it cannot be reused to copy over diverse hardware components.
- Also, it needs to take care of all sorts of error conditions in the keyborad and printer component (lots of unncessary details creep in).

DIP: Summary

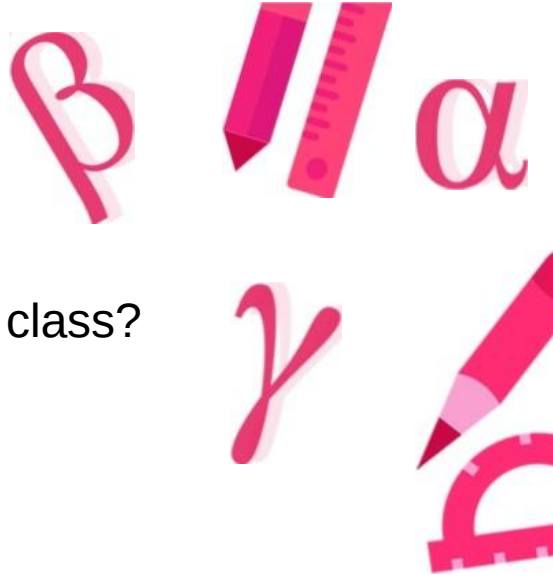
- DIP promises many benefits of OO paradigm.
- Reusability is greately enhanced by DIP.
- Code can be made resilient to change by using DIP.
- As a result, code is easier to maintain.

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.



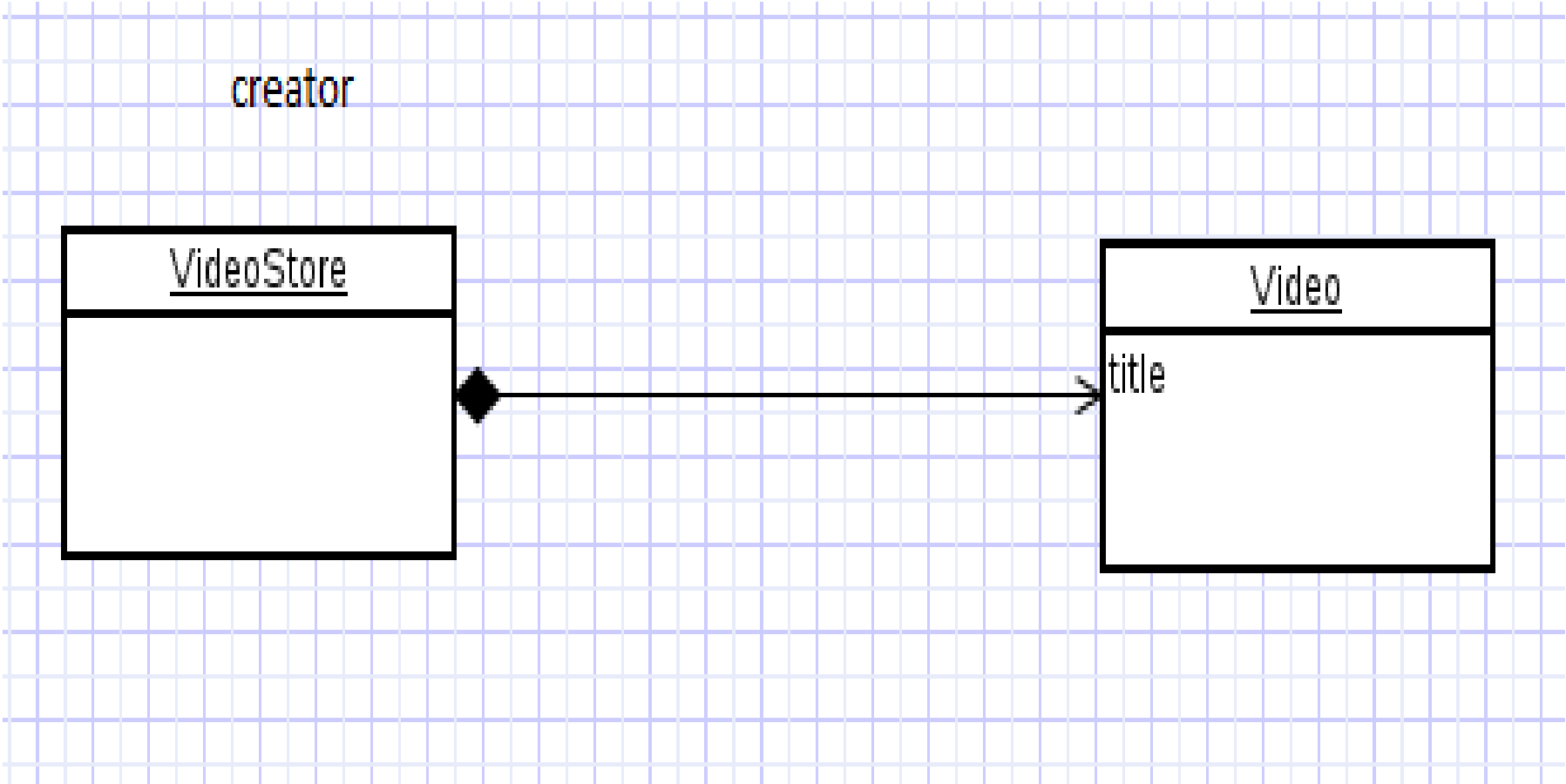
Creator

- 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.

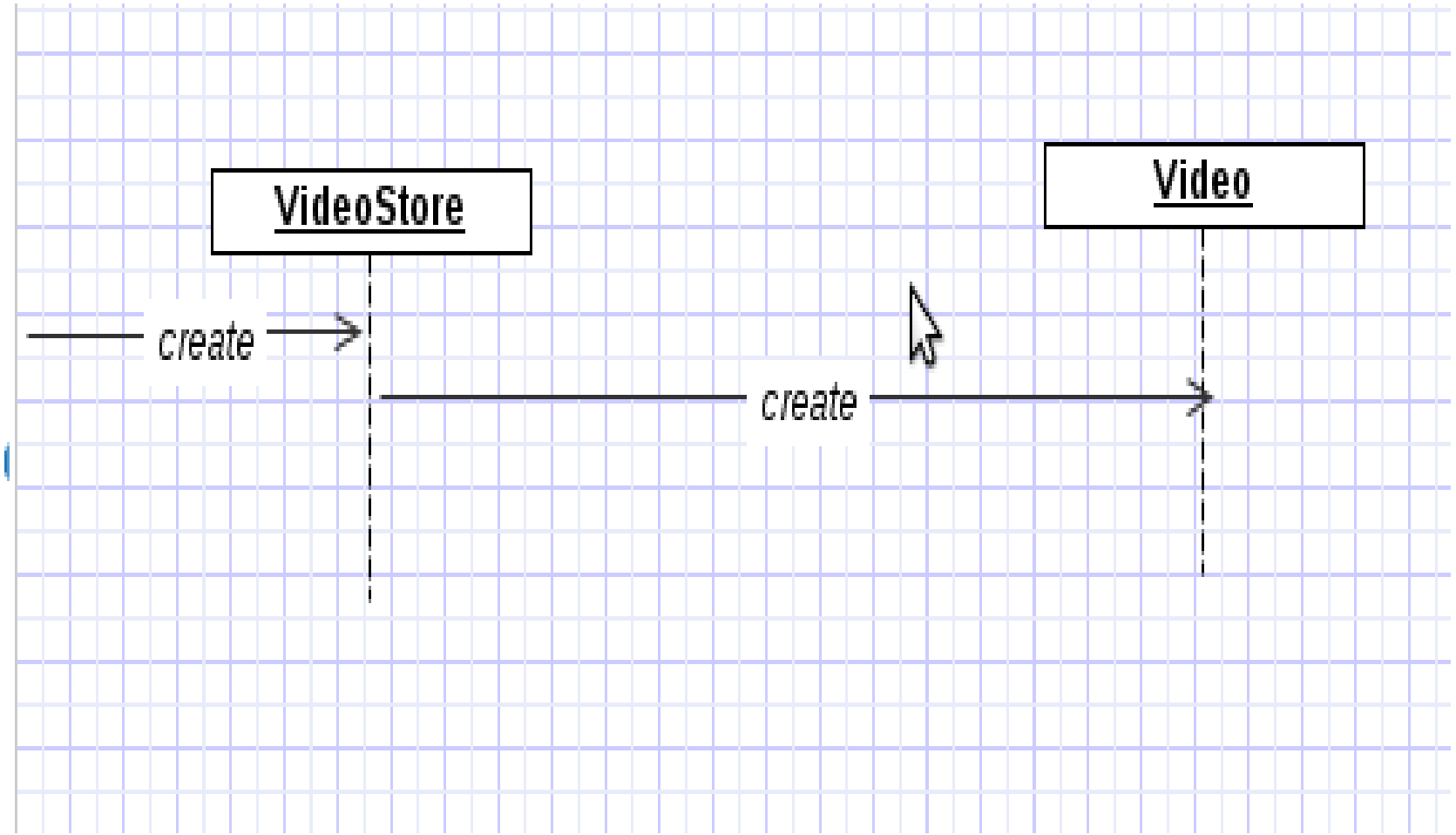
Example for Creator

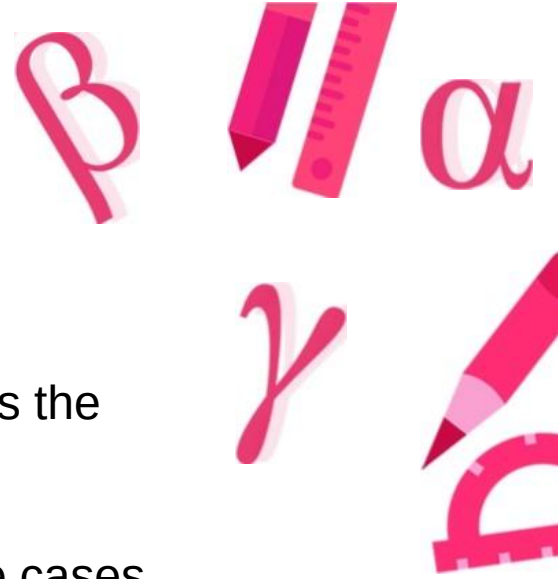
- Consider VideoStore and Video in that store.
- VideoStore has an aggregation association with Video. I.e, VideoStore is the container and the Video is the contained object.
- So, we can instantiate video object in VideoStore class

Example diagram



Example for creator





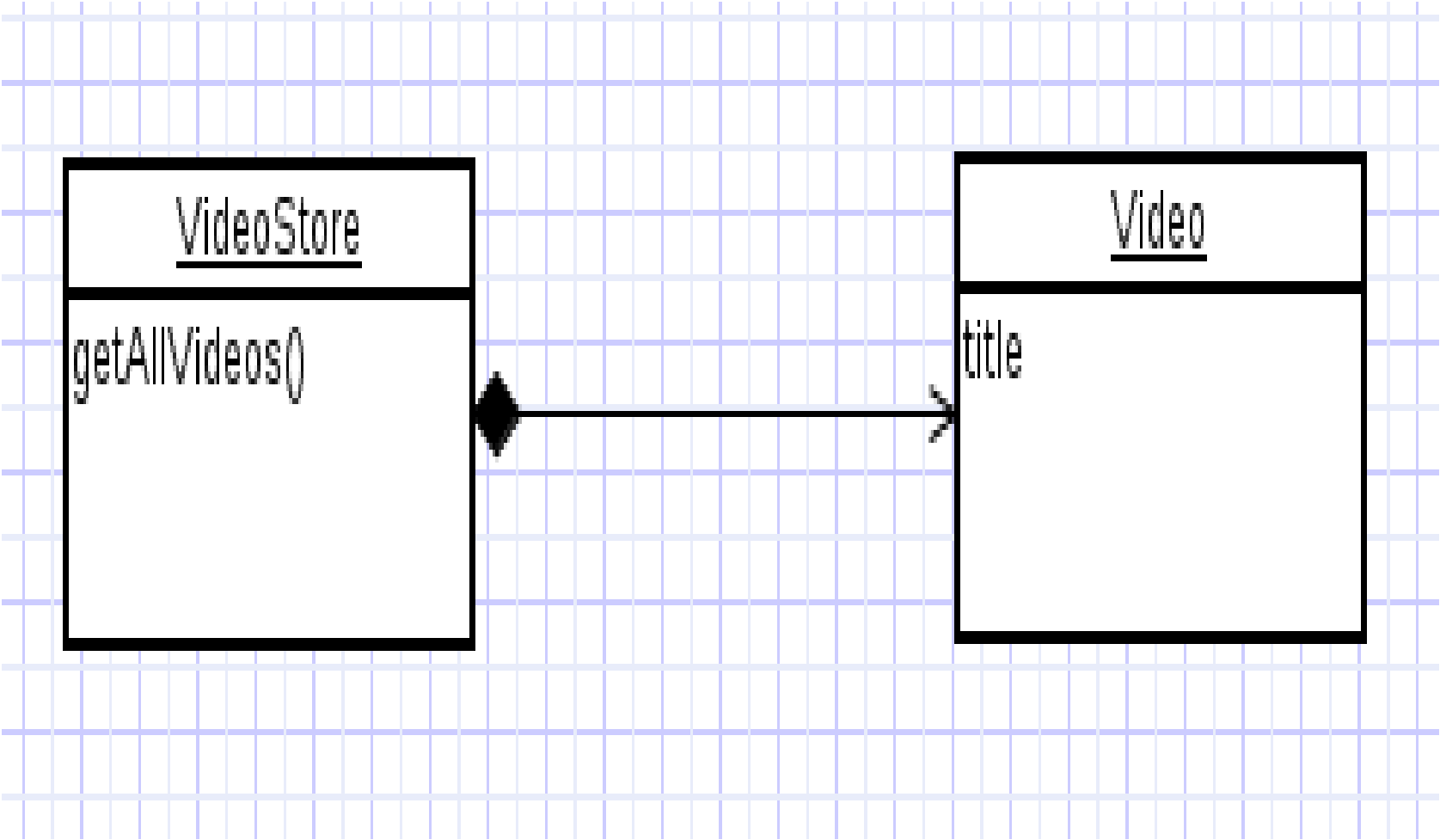
Expert

- 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.

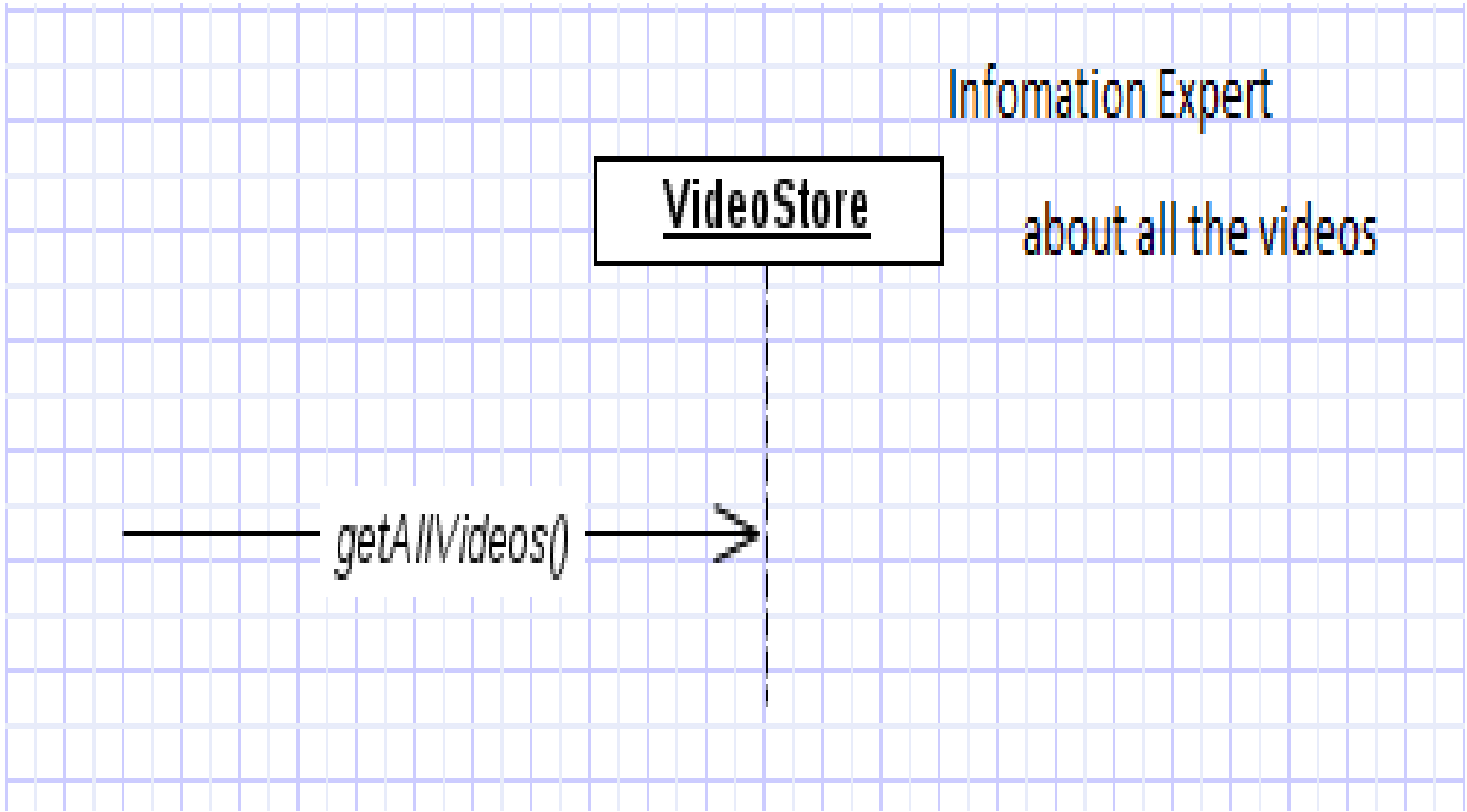
Example for Expert

- 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



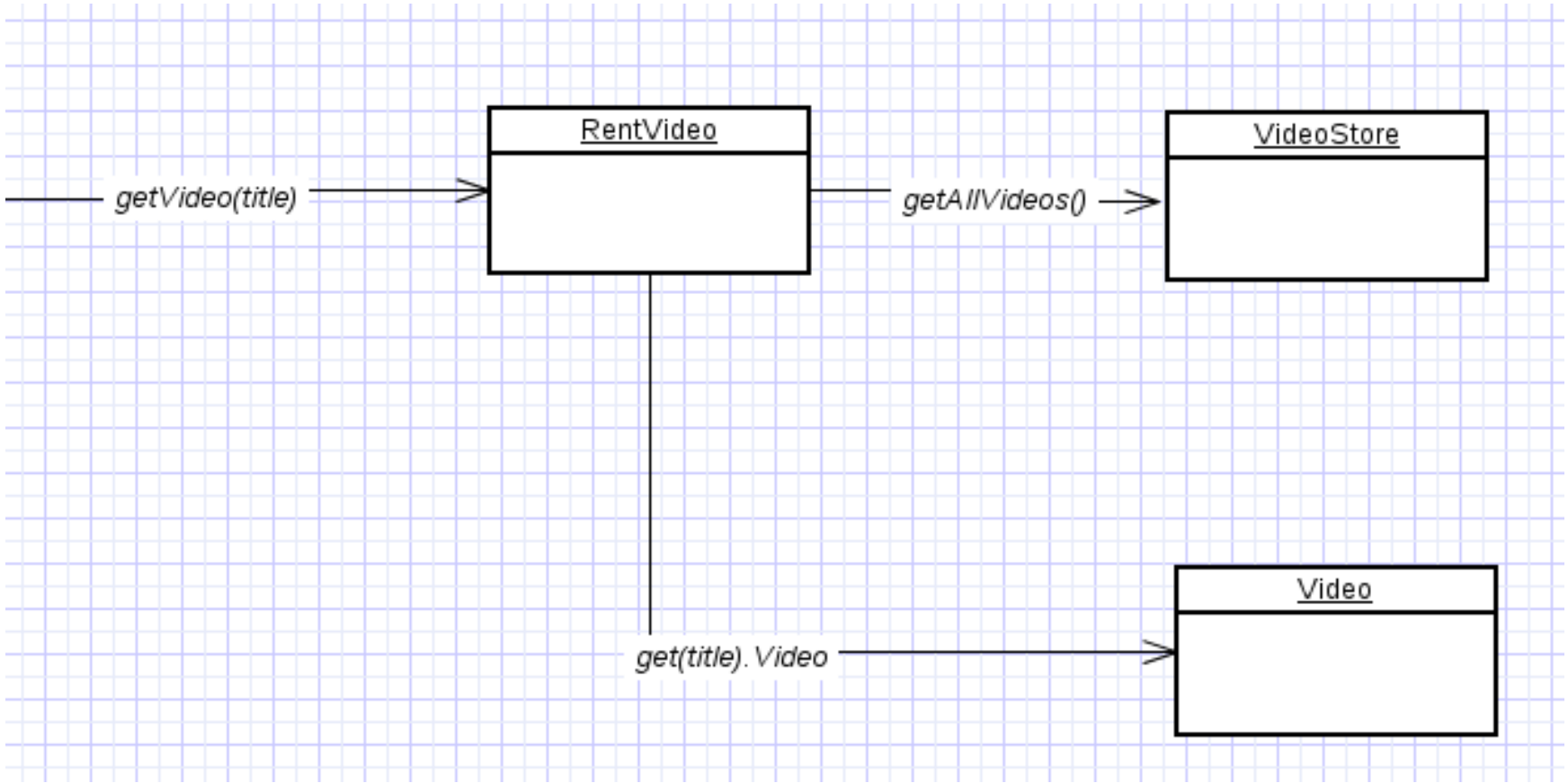
Example for Expert



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

Example for poor coupling

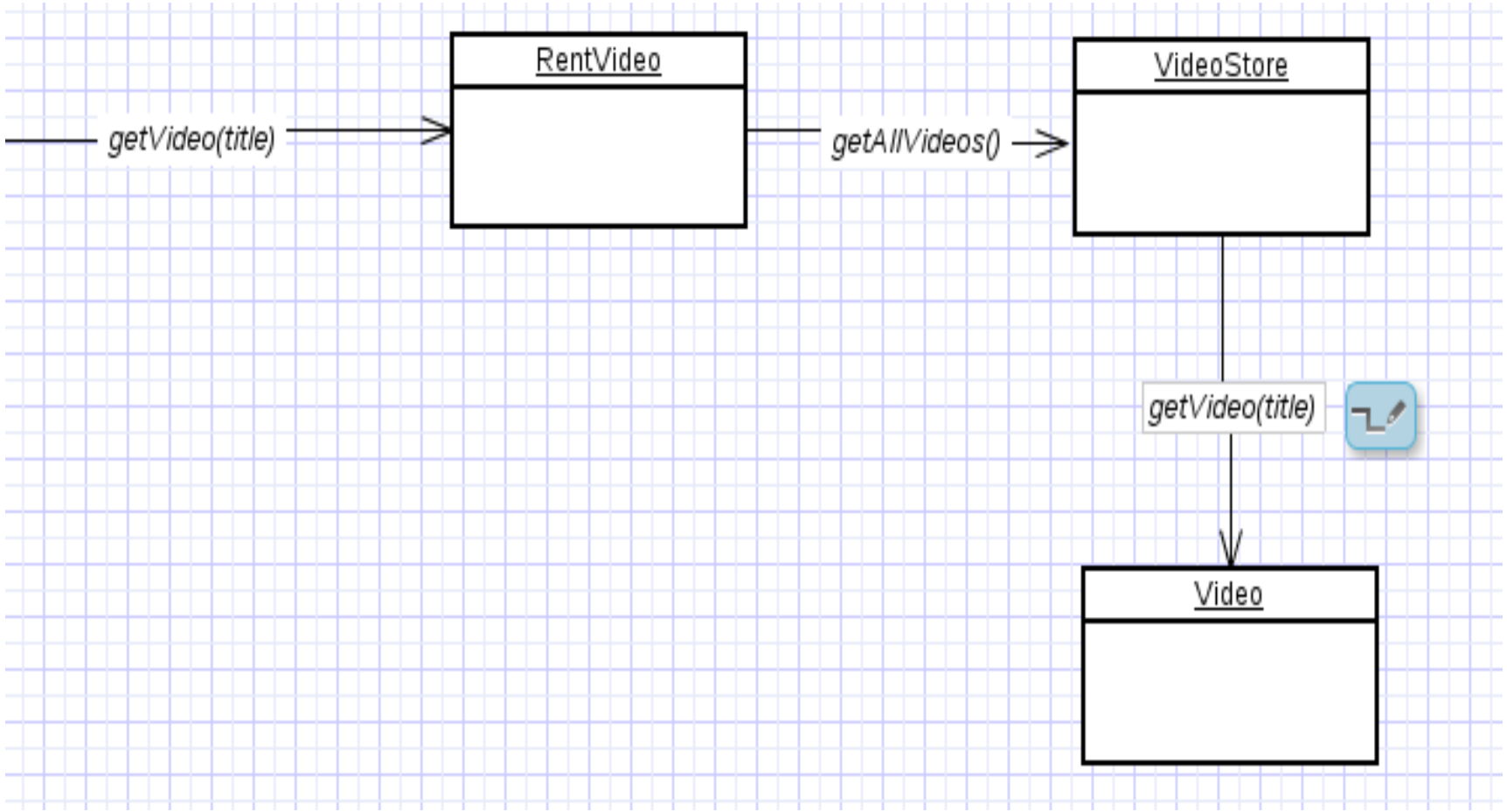


here class Rent knows about both VideoStore and Video objects. Rent is depending on both the classes.

Example for low coupling

- VideoStore and Video class are coupled, and Rent is coupled with VideoStore.

Thus providing low coupling.



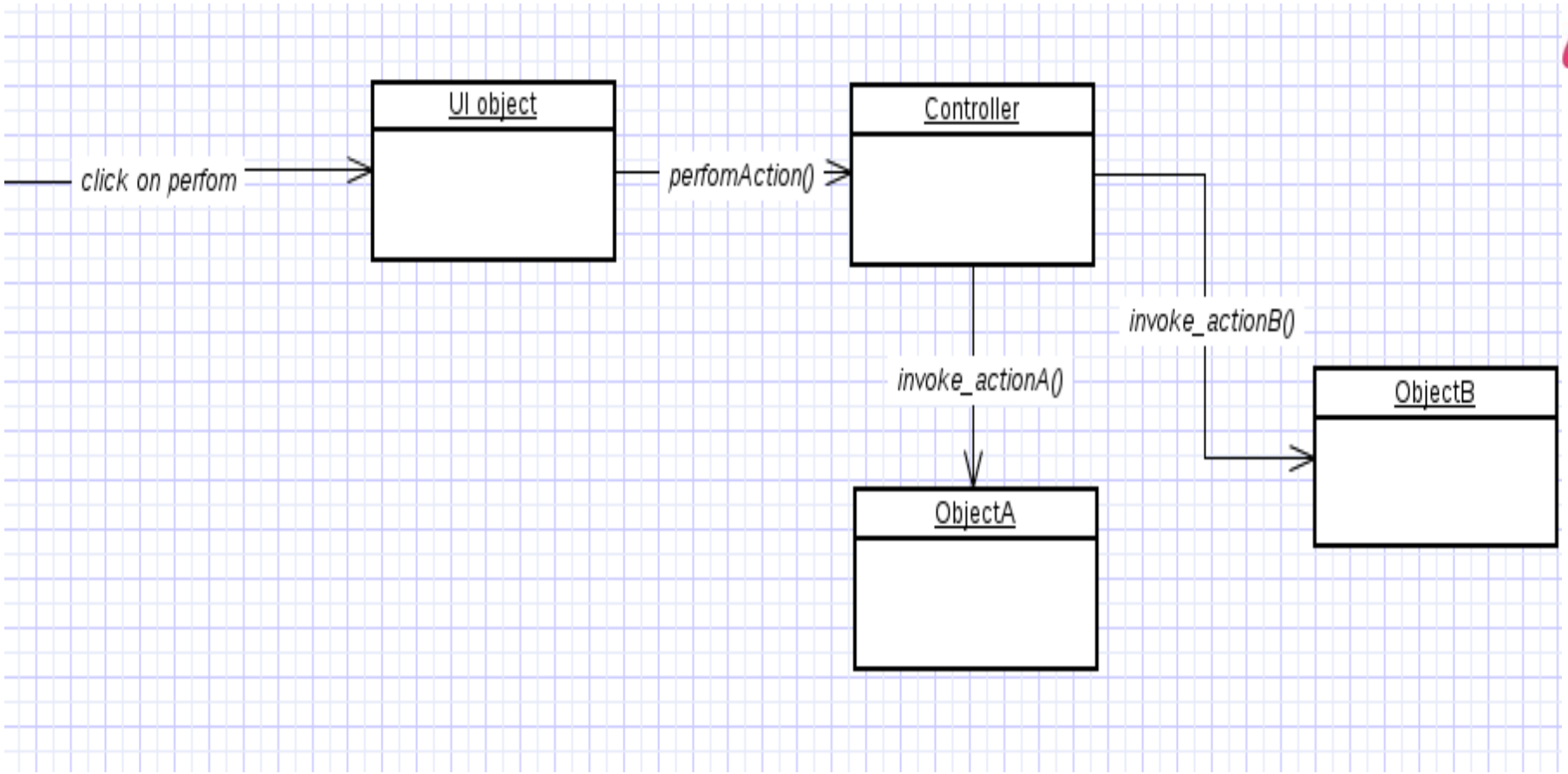
Controller

- 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

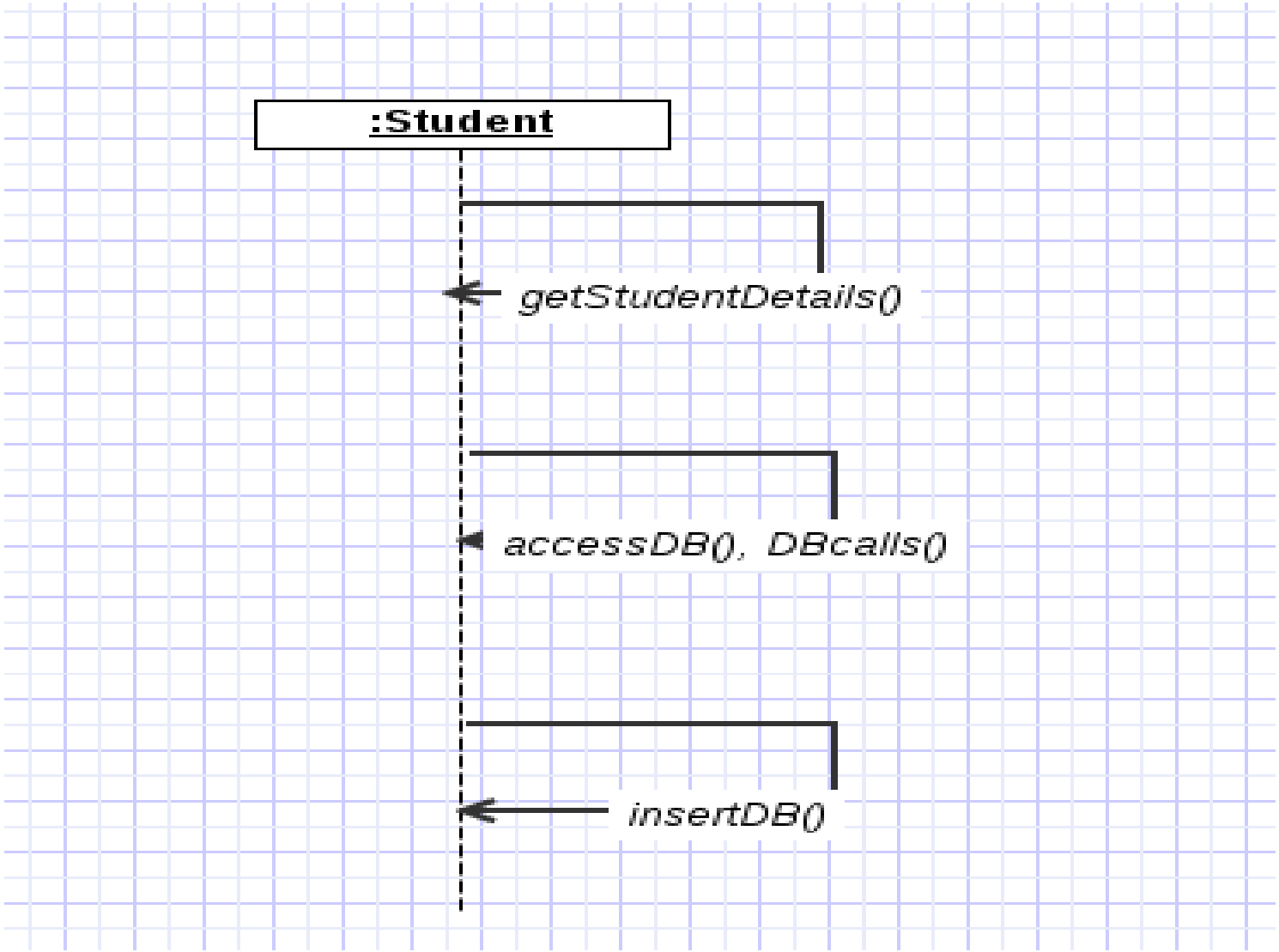
  - Controller class also performing many tasks instead of delegating to other class.

Solution – controller class has to delegate things to others.

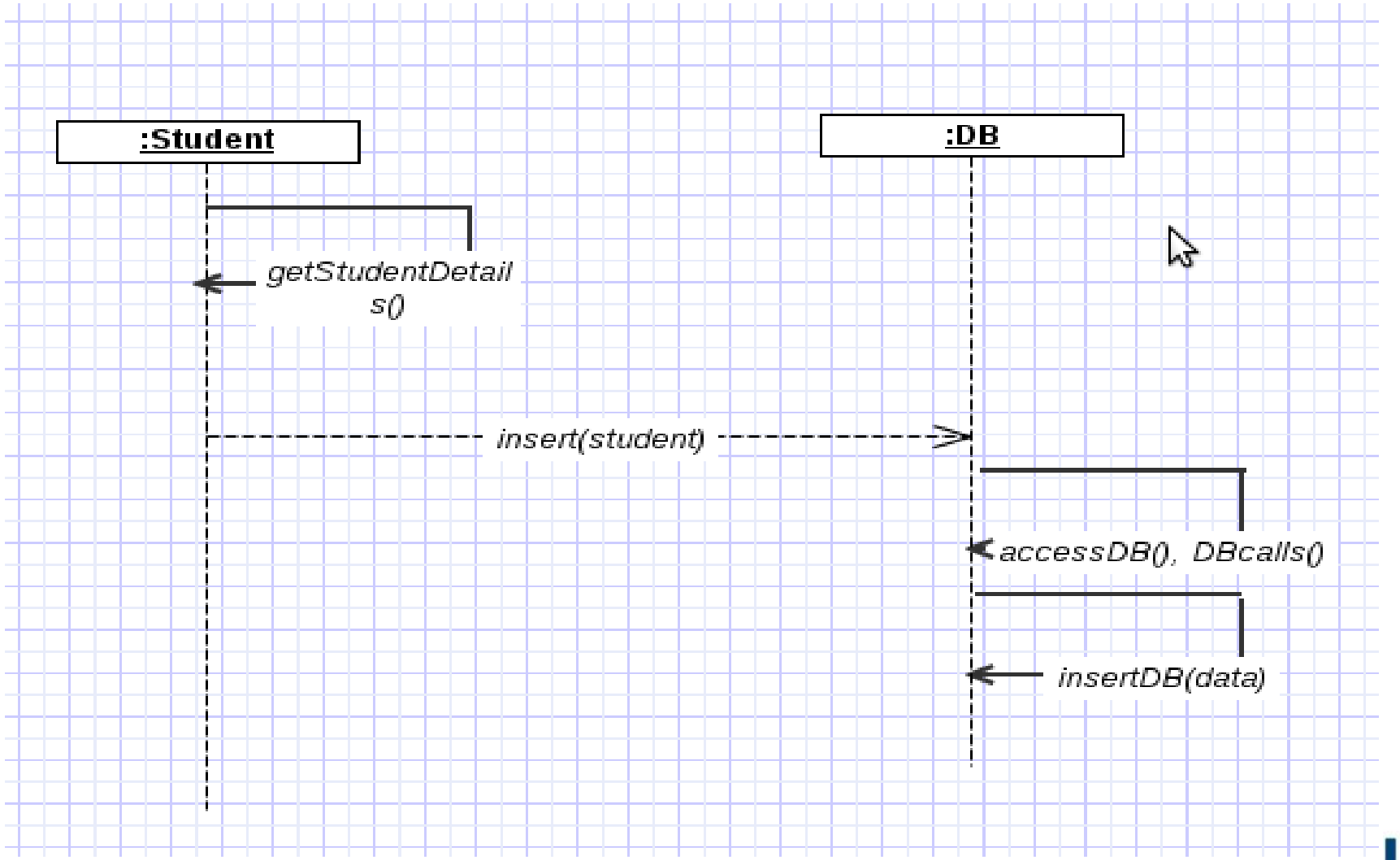
High Cohesion

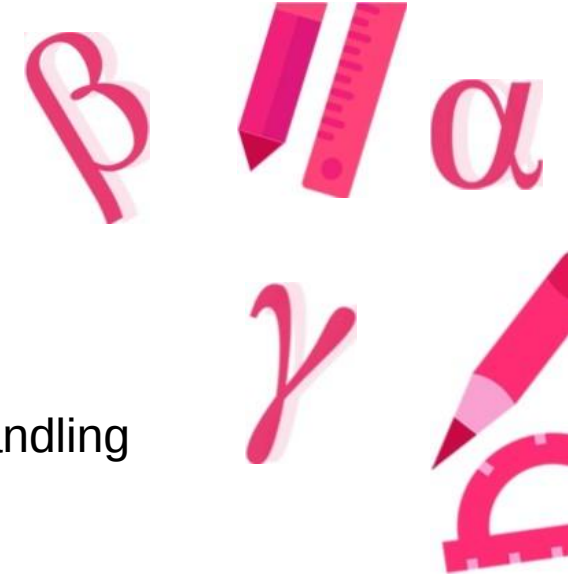
- How are the operations of any element are functionally related?
- Related responsibilities in to one manageable unit.
- Prefer high cohesion
- Clearly defines the purpose of the element
- Benefits
- Easily understandable and maintainable.
- Code reuse
- Low coupling

Example for low cohesion



Example for High Cohesion



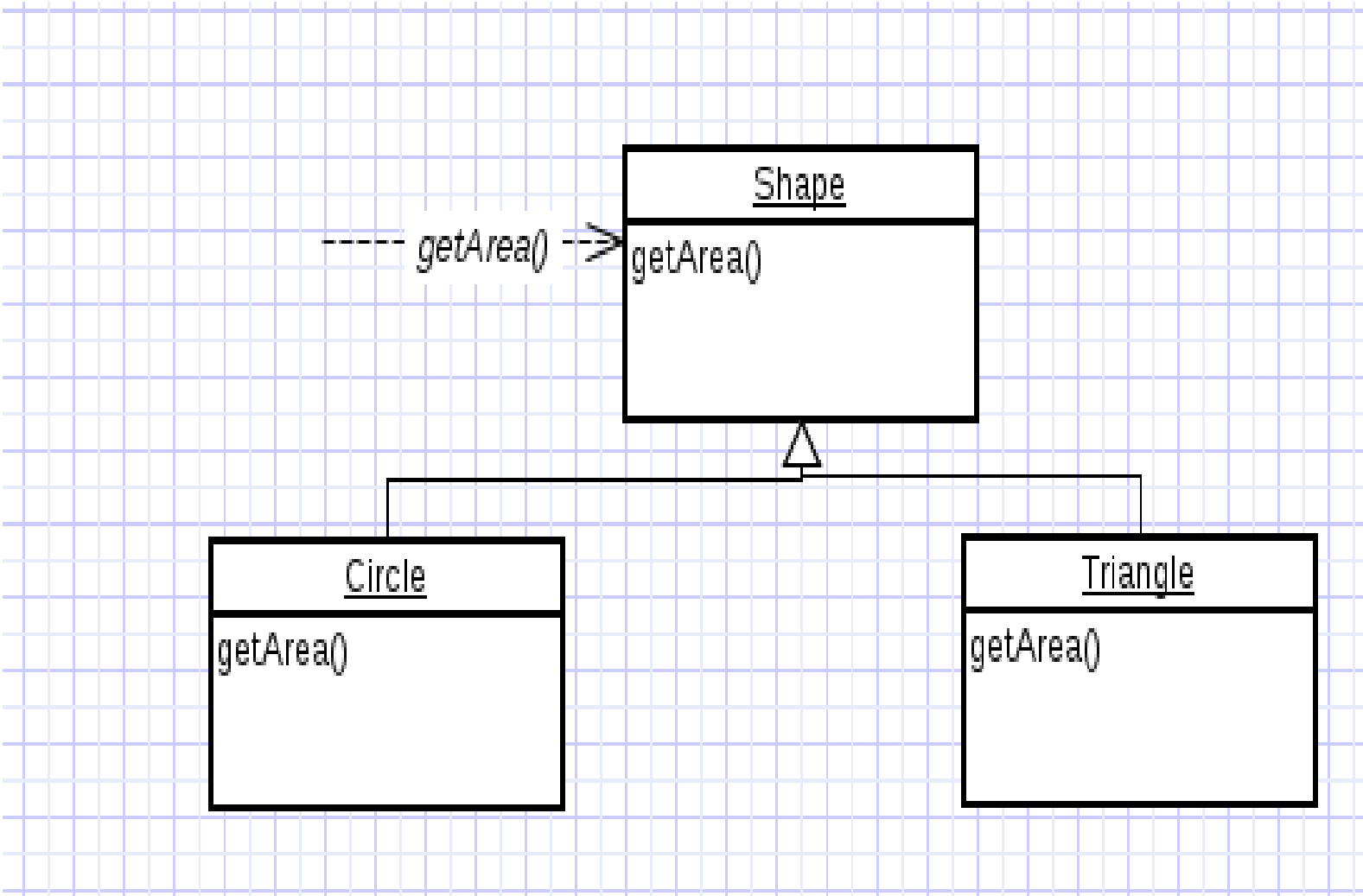


Polymorphism

- 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.


Shape
getArea()
getArea()
Circle
getArea()
Triangle
getArea()

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

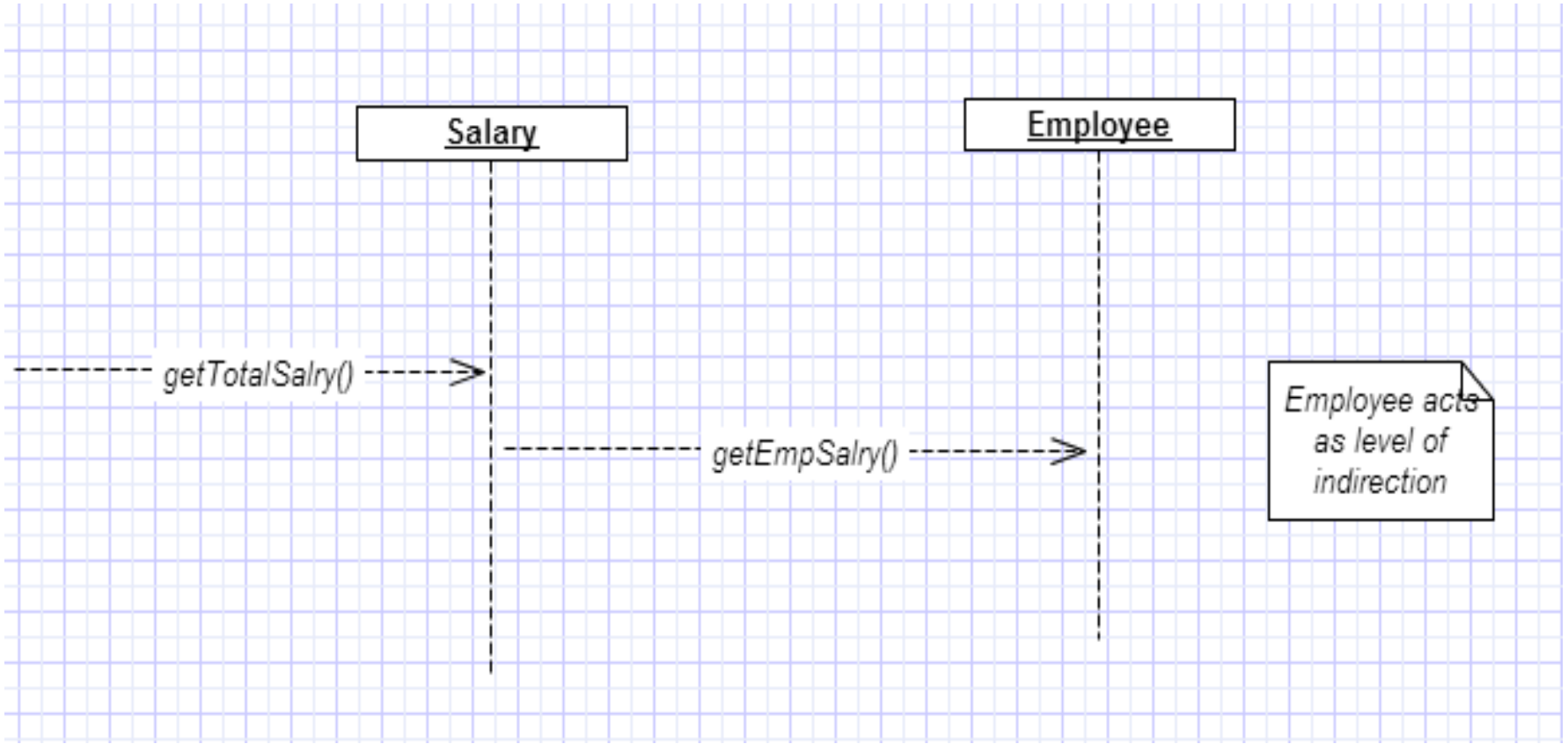
- Suppose we Shape class, if we must store the shape data in a database.
- If we put this responsibility in Shape class, there will be many database related operations thus making Shape incohesive.
- So, create a fabricated class DBStore which is responsible to perform all database operations.
- Similarly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

Thank You!