Object Oriented Analysis and Design

Unit 4 :- GRASP

GRASP

- General Responsibility Assignment Software Patterns
- Assigning responsibilities to collaborating objects
- Responsibility can be accomplished by a single object or by a group of objects
- **Patterns**
  - Creator
  - Information Expert
  - Low Coupling
  - High Cohesion
  - Controller
  - Indirection
  - Polymorphism
  - Protected Variations
  - Pure Fabrication

Creator

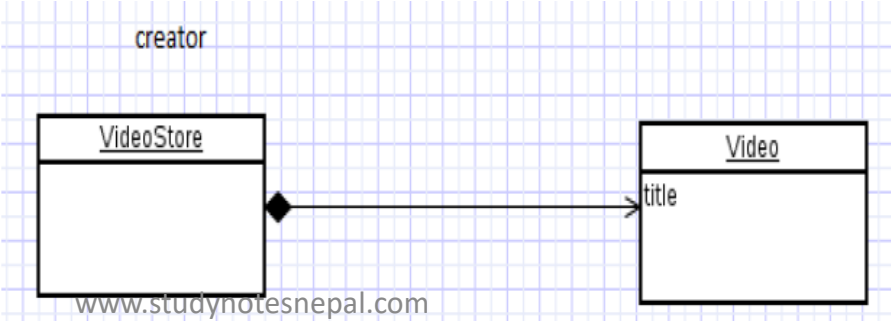
- Who creates an Object? Or who should create a new instance of some class?
- “Container” object creates “contained” objects.
- Decide who can be creator based on the objects association and their interaction

Creator…

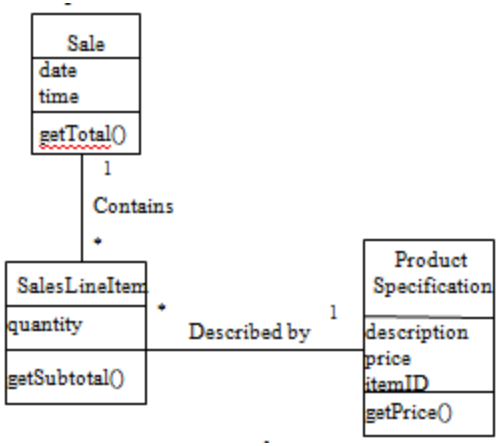
- Problem: Who is responsible for creating new instances of some class?
- Solution: Assign class B the responsibility to create an instance of class A if one or more of the following is true:
  - B *aggregates* A
  - B *contains* A (composition)
  - B *records* instances of A objects
  - B *uses* A objects
  - B *has the initializing data* that will be passed to A when it is created

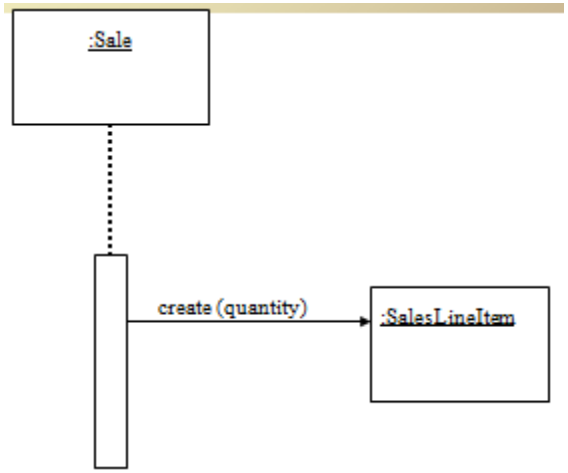
Creator (Example)…

- 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



Creator (Example)…





Abbreviated Sequence Diagram

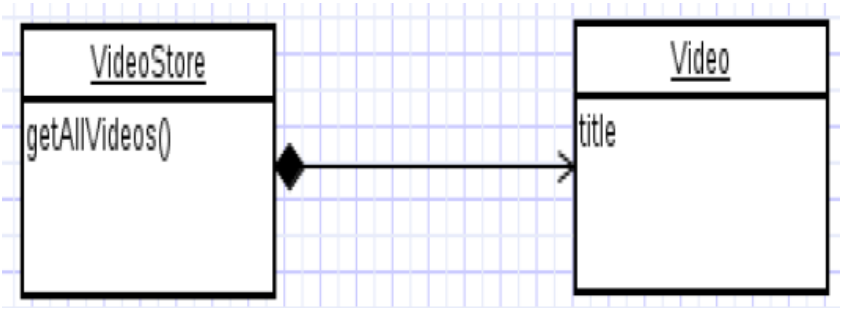
- Who should be responsible for creating a ***SalesLineItem*** instance?
- Since a ***Sale*** contains many ***SalesLineItem*** objects, the Creator pattern suggests that ***Sale*** is a good candidate to have the responsibility of creating ***SalesLineItem*** objects
- i.e. Sales contains / has_a (one or more) SalesLineItems {Aggregations}

Information 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

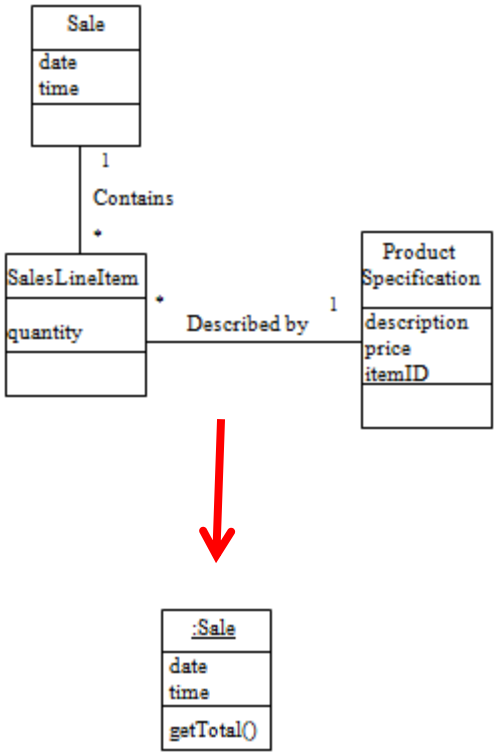
Information Expert (Example)…

- 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.



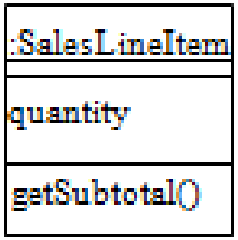
Information Expert (Example)…

- Who should be responsible for knowing the grand total of a sale?
- i.e. 'who' has the information needed to determine the total?
- Look in domain model (***Sale***)
- In ***'Sale'*** class give the responsibility of knowing its total, expressed with a method named ***getTotal***



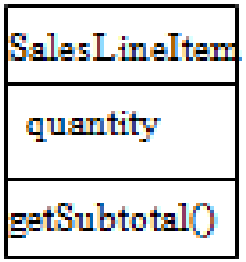
Information Expert (Example)…

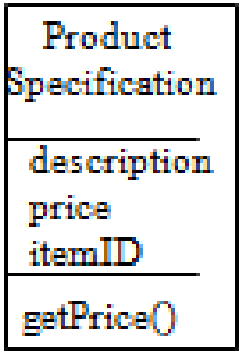
- What information is needed to determine the line item subtotals?
- We need:
  - ***SalesLineItem.quantity*** and ***ProductSpecification.price***
- The ***SalesLineItem*** "knows" its quantity (is typically an attribute) and its associated ***ProductSpecification*** **(**via association);
- Therefore, by Expert, ***SalesLineItem*** should determine the *subtotal*; it is the *Information Expert* in this case.
- So, 'now' what do we have?



Information Expert (Example)…

- To fulfill the responsibility of knowing and answering the *subtotal*, a ***SalesLineItem*** needed to know the product price.
- The ***ProductSpecification*** is also an information expert on answering its price, thus we need a message sent to ***ProductSpecification*** asking for the price, (something like: ***price*** and ***getPrice()***)





Information Expert…

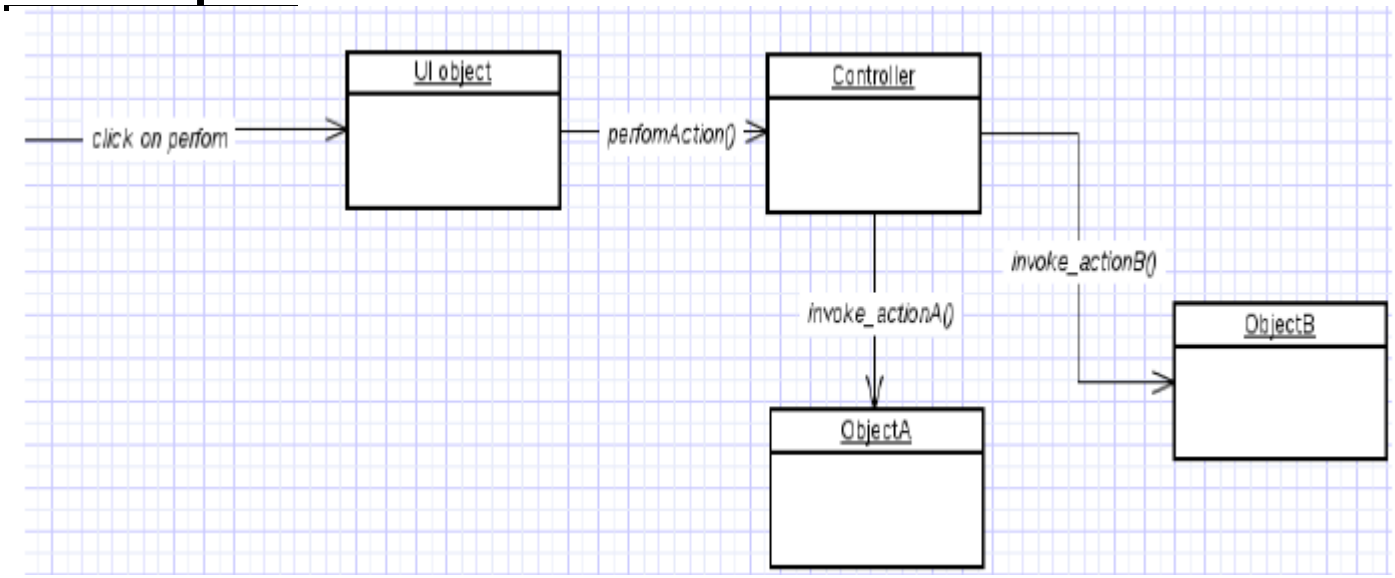
- Be careful: Don't run 'Expert' into the Ground
- Who should be responsible for saving ***Sale*** in a database?
- Always want to separate I/O from computations
- Each class would have its own services to save itself in the database.
- Sale would have to now contain logic related to database handling, such as related to SQL and JDBC (for J2EE) or more

Controller Pattern

- 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 Pattern…

- Example

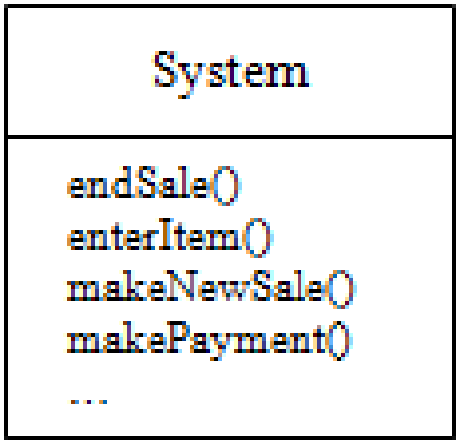


Controller Pattern…

- Useful in developing web applications
- Who should be responsible for handling an input system “event”?
- System Event – event generated by external actor
  - Eg :- Clicking “***End Sale***” Button
- A Controller is a non-user interface object responsible for receiving or handling a system event
- Input events might come from
  - a GUI operated by a person, or
  - a call from a telecommunications switch, or
  - a signal from a sensor, etc

Controller Pattern…

- Do not infer that there will be a class named **System** in Design.
- Rather, during Design, a **Controller** class is assigned the responsibilities for system operations



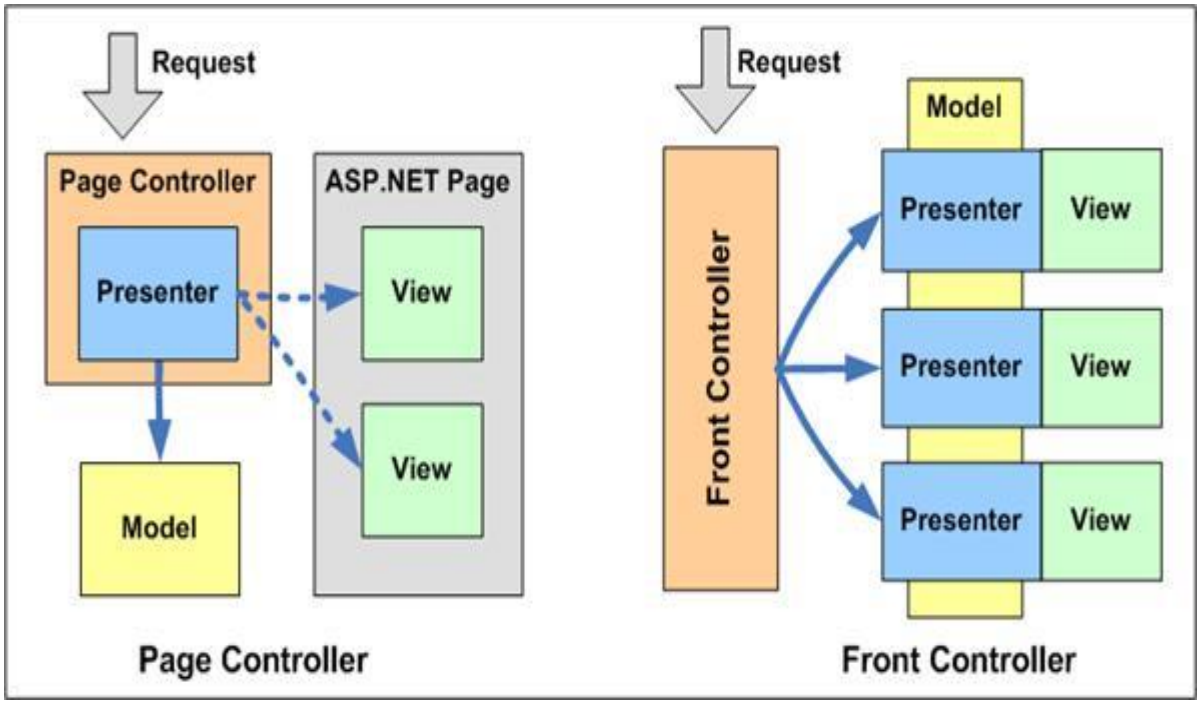
Controller Pattern…

- Do not give controllers too much responsibility (***bloated controller***)
- Solution $\rightarrow$ add more controller
- Types
  - Façade controller $\rightarrow$ represents the overall system
  - Session (use case) controller $\rightarrow$ represents a use case
- Façade Controllers are used when there are not too many events to control

Controller Pattern…

- Other types
  - ***Page Controller***
    - Controller that uses a single Presenter which interacts with the Model (the data for the page). When it receives a request, the Page Controller can determine which partial View to display within the page, and then interact with that View following the MVP pattern
  - ***Front Controller***
    - A separate controller that examines each request and determines which page to display. Each page is a complete implementation of MVC, with its own View, and each Presenter interacts with the View and the Model (the data)

Controller Pattern…



Low Coupling

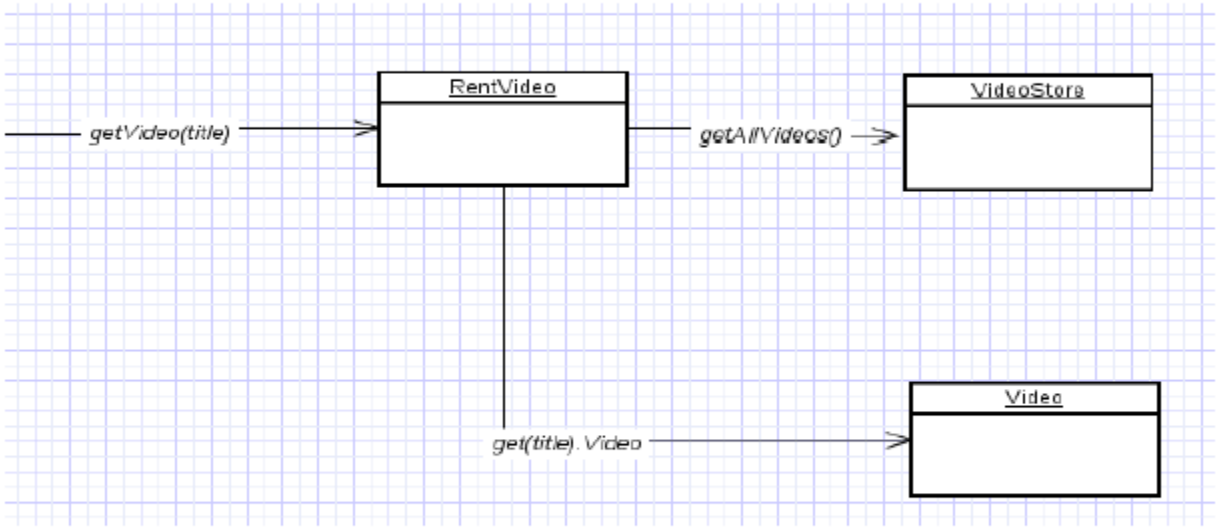
- How strongly the objects are connected to each other?
- ***Coupling*** – object depending on other object.
- When depended upon element changes, it affects the dependant also.
- ***Low Coupling*** – How can we reduce the impact of change in depended upon elements on dependant elements.
- Prefer low 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.

Low Coupling…

- Coupling is a measure of how strongly one element is connected to another element
  - Common forms of coupling
    - Class A has an instance of Class B
    - Class A send a message to an instance of Class B
    - Class A is a subclass of Class B (inheritance)
    - Class A implements interface I
- Highly coupled classes suffer from following problems
  - Forced local changes because of changes in related classes
  - Harder to understand in isolation
- Solution
  - Assign the responsibility so that coupling remains low

Low Coupling…

Example for poor coupling

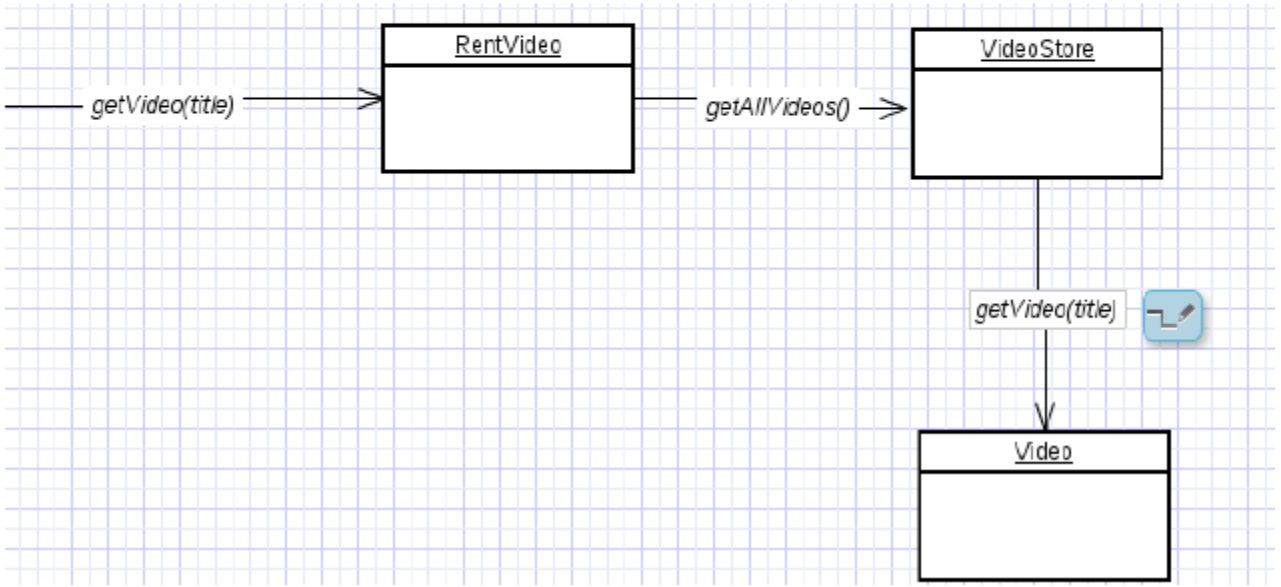


here class Rent knows about both VideoStore and Video objects. Rent is depending on both the classes.

Low Coupling…

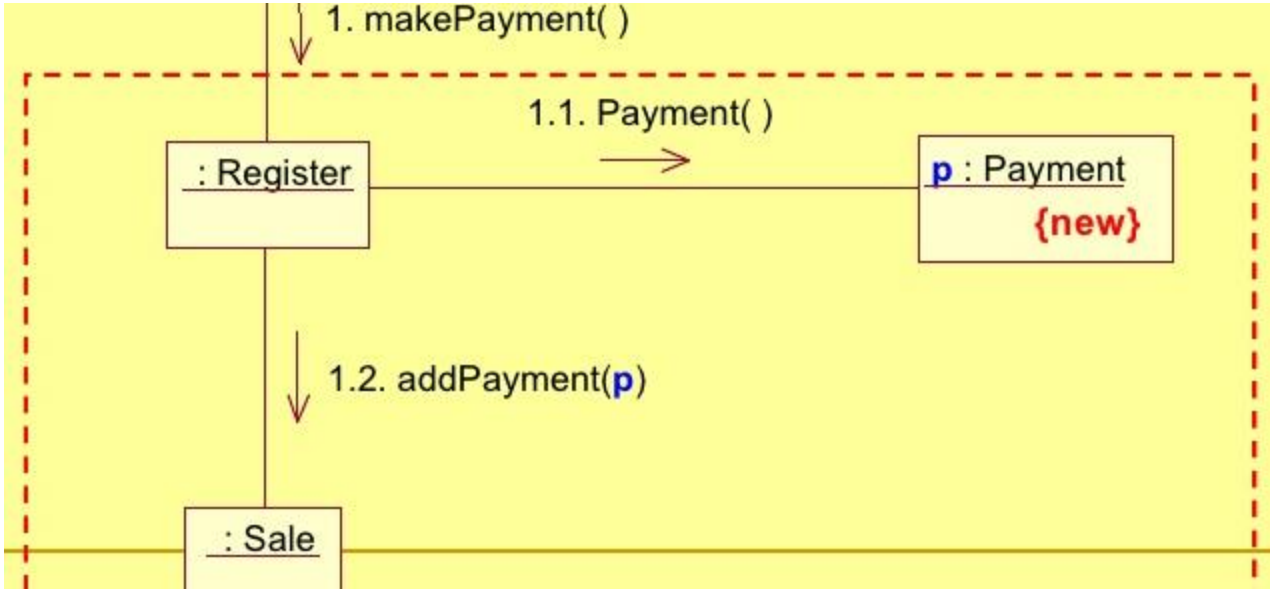
Example for low coupling

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



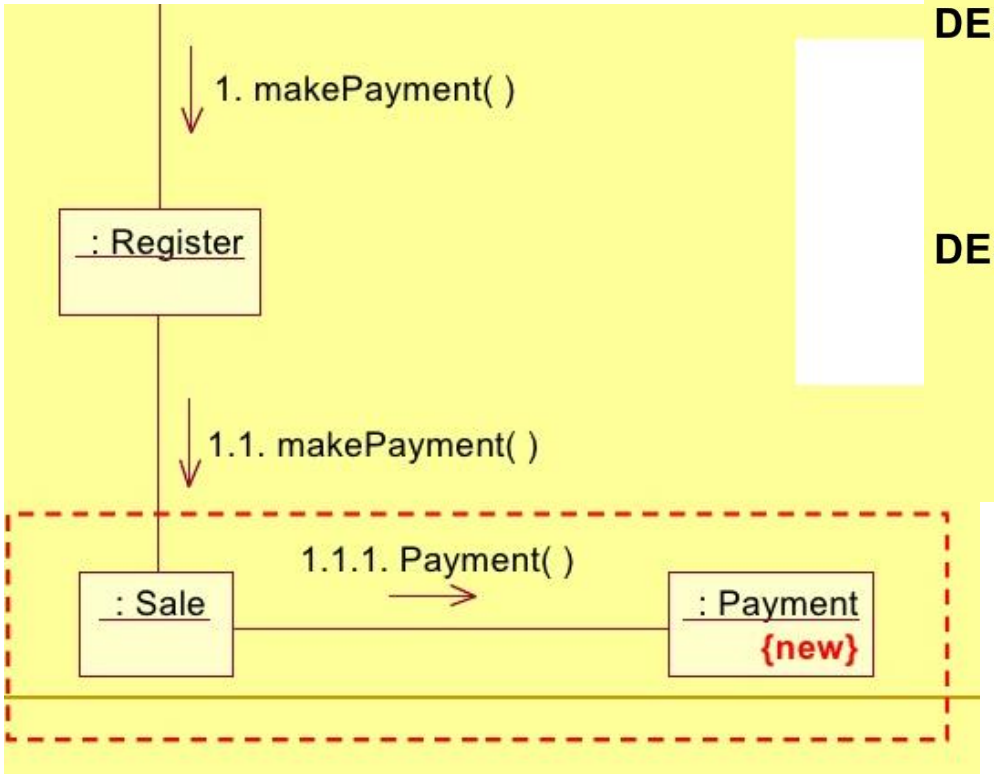
Low Coupling…

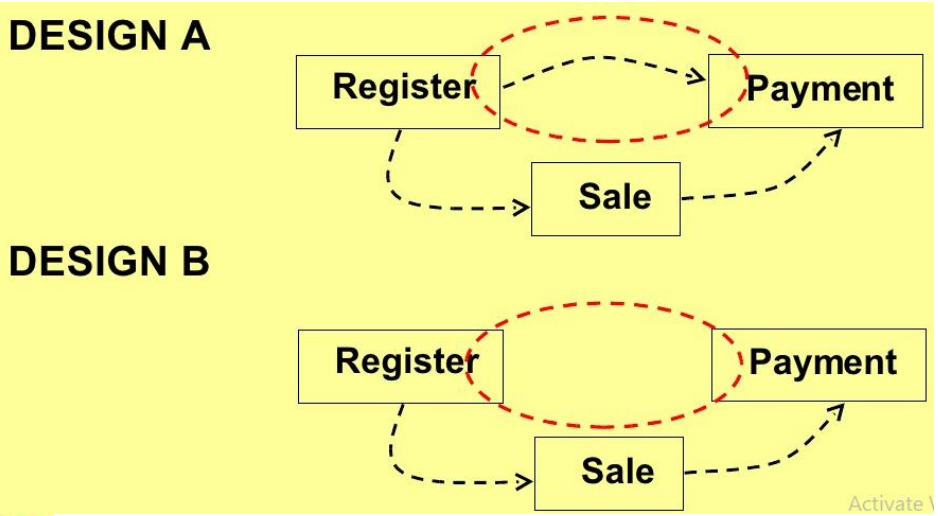
- Example
  - In POS application, when the cashier enters a payment, payment object needs to be created and associated with the current sale
  - In real world domain, a Register records a Payment, so Register will be creator for that responsibility



Low Coupling…

- Example
  - Since the Payment object will eventually be linked to the Sale object, why not assign the responsibility to Sale?



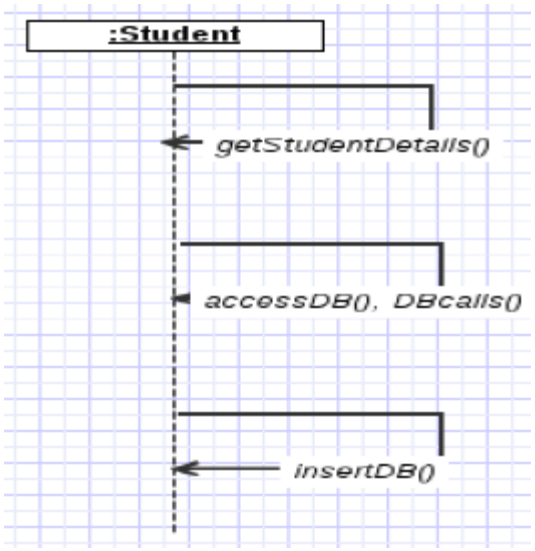


Based on the coupling factor we prefer design B

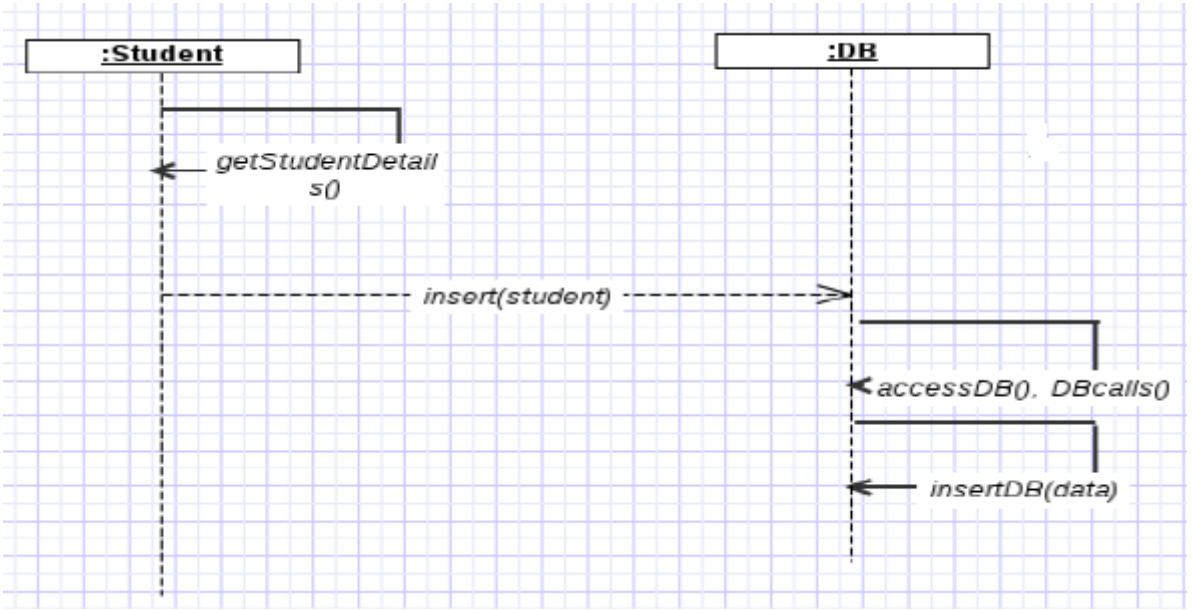
High Cohesion

- How are the operations of any element are functionally related?
- Related responsibilities in to one manageable unit.
- High cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module
- Prefer high cohesion
- Clearly defines the purpose of the element
- Benefits
  - Easily understandable and maintainable.
  - Code reuse
  - Low coupling

High Cohesion…



Low Cohesion



High Cohesion

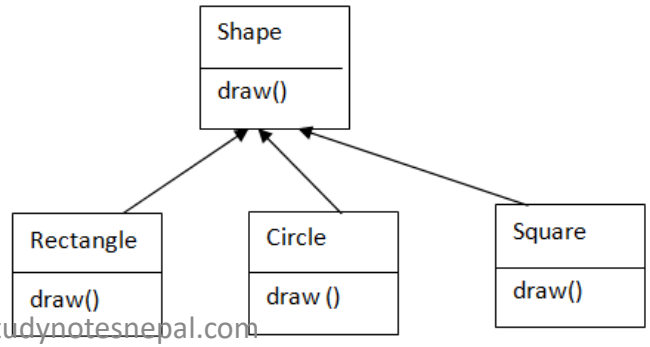
High Cohesion…

```
 // Less cohesive class design

class BudgetReport {

    void connectToRDBMS() {

    }
    void generateBudgetReport() {
    }
    void saveToFile() {
    }
    void print() {
    }
}
```

```
More cohesive class design
// More cohesive class design
class BudgetReport {
    Options getReportingOptions() {
    }
    void generateBudgetReport(Options o) {
    }
}
class ConnectToRDBMS {
    DBconnection getRDBMS() {
    }
}
class PrintStuff {
    PrintOptions getPrintOptions() {
    }
}
class FileSaver {
    SaveOptions getFileSaveOptions() {
    }
}
```

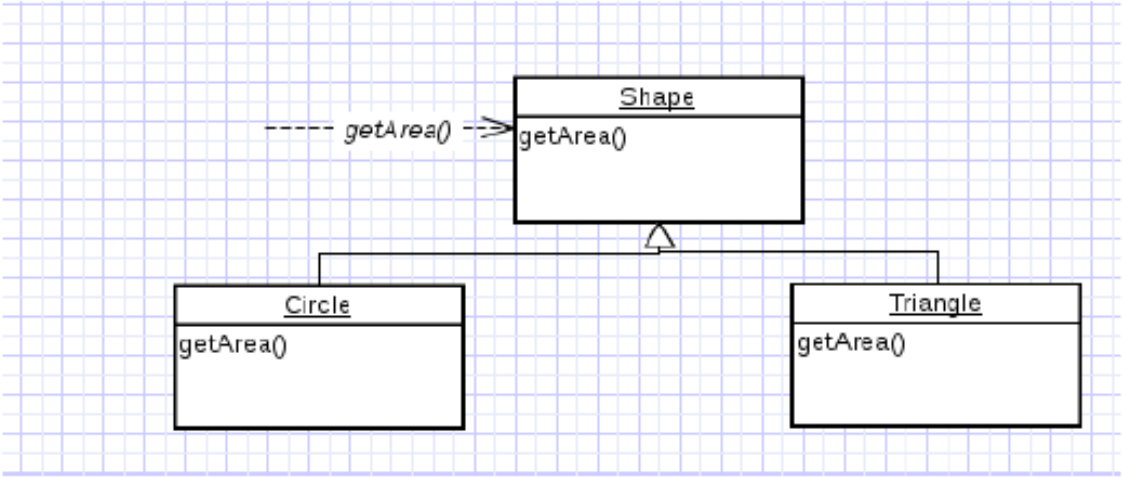
Polymorphism

- 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.



Polymorphism(Example)…

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



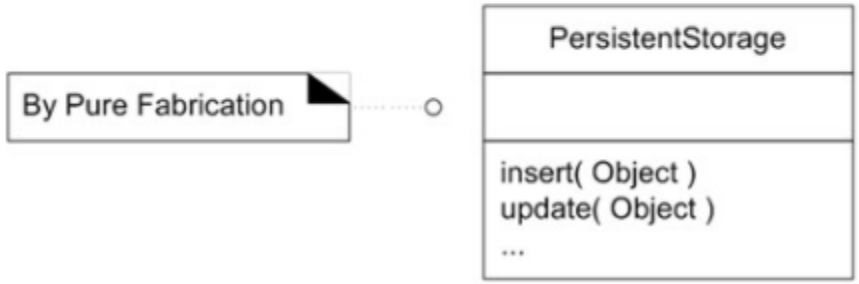
- By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle.

Pure Fabrication

- Fabricated class/ artificial class – assign set of related responsibilities that doesn't represent any domain object.
- i.e. a pure fabrication is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling and high cohesion
- Provides a highly cohesive set of activities.
- Behavioral decomposed – implements some algorithm.
- Examples: Adapter
- Benefits: High cohesion, low coupling and can reuse this class.

Pure Fabrication (Example)…

- **Example:** Suppose we need to save **Sale** object in a relational DB.
- Information Expert or Expert says Sale should do it, because Sale knows its total.
- But it violates Low Coupling and High Cohesion because Sale will be coupled with JDBC etc.
- ***Sale*** remains well-designed, with high cohesion and low coupling
- The ***PersistentStorage*** class is itself relatively cohesive, having the sole purpose of storing or inserting objects in a persistent storage medium
- The ***PersistentStorage*** class is very generic and reusable object



Pure Fabrication (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

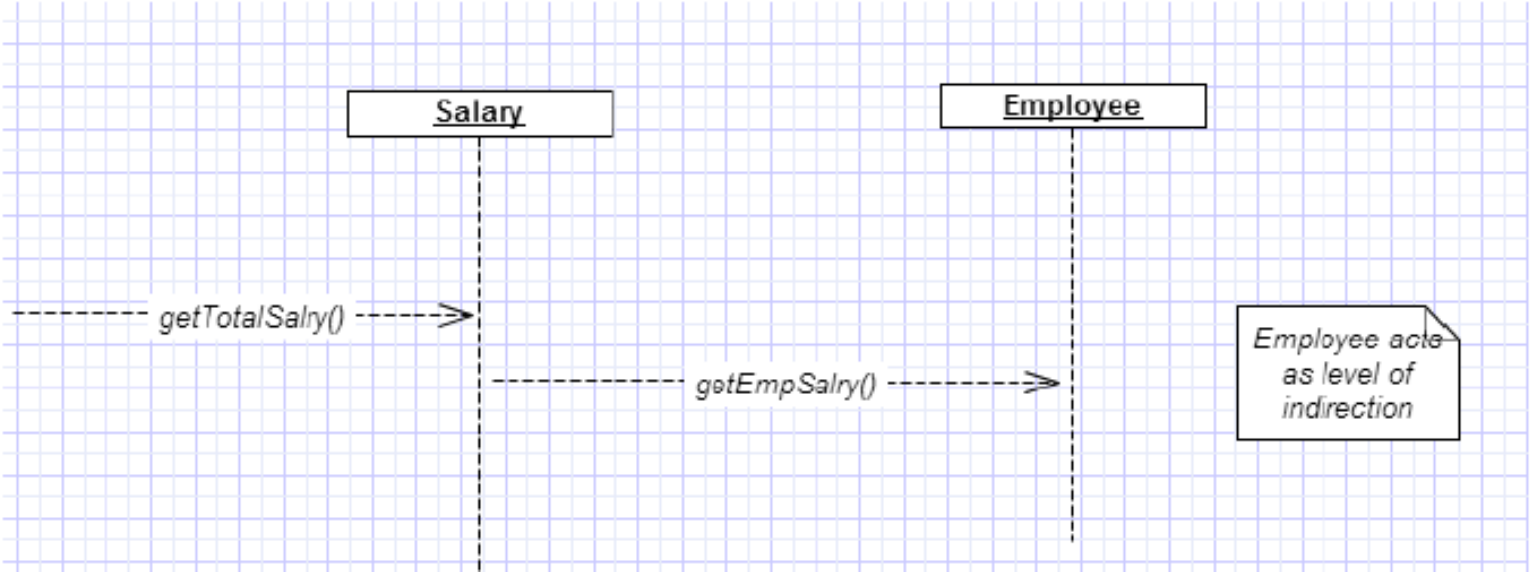
- How can we avoid a direct coupling between two or more elements.
- Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.
- Benefits: low coupling
- Example: Adapter, Obserever

Indirection…

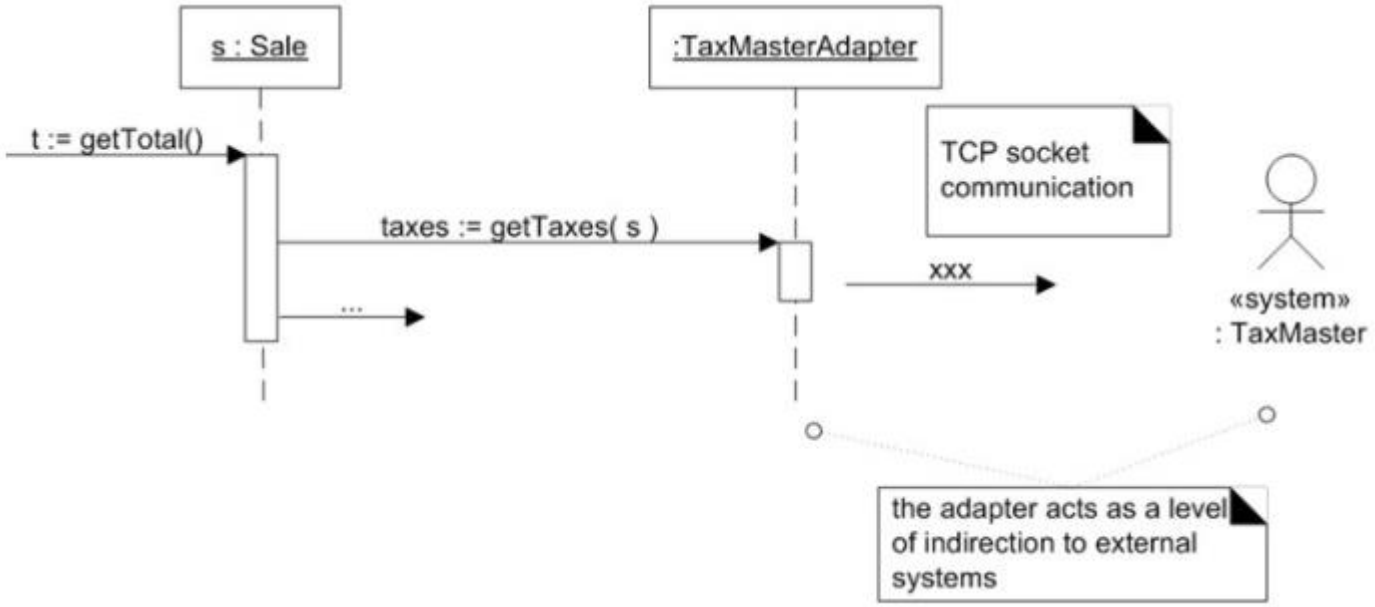
- **Context/Problem:** Where to assign responsibility, to avoid direct coupling between two (or more) things? How to decouple objects so that low coupling is supported and reuse remains high?
- **Solution:** Create an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates and *indirection* between the other components.

Indirection(Example)…

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



Indirection(Example)…



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: data encapsulation, interfaces

Protected Variation…

- **Context/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.
- **Example:** Prior external tax calculator is also a protected variation example.

GoF Design Pattern

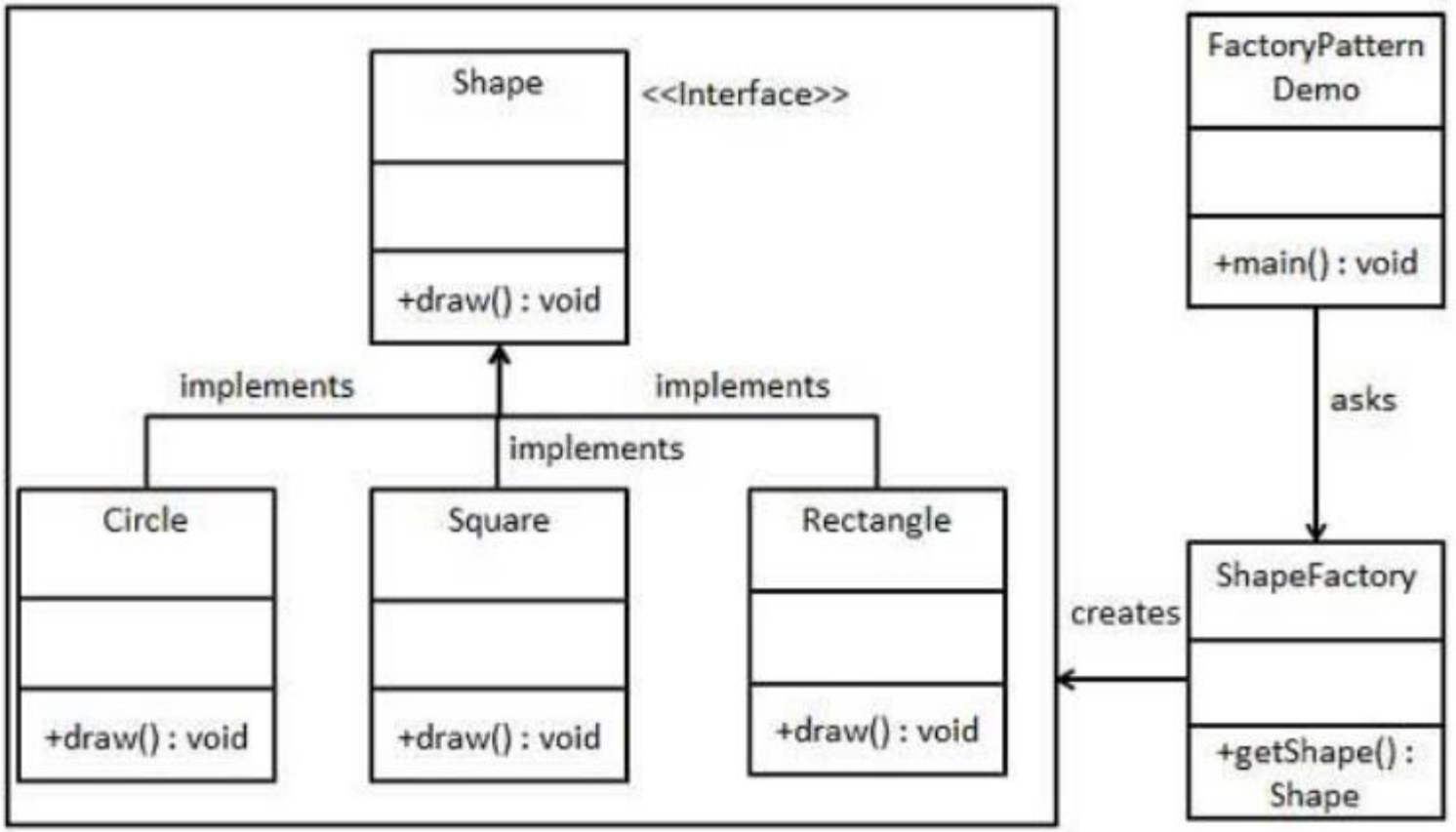
- Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides)
- Adapter, Singleton, Factory, Observer

Factory Pattern

- Problem
  - Who should be responsible for creating objects when there are special considerations such as complex creation logic, a desire to separate creation responsibilities for better cohesion etc
- Solution
  - Create pure fabrication object called a Factory that handles the creation

Factory Pattern…

- In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface



Factory Pattern..

Step 1

Create an interface.

Shape.java

```
public interface Shape {

   void draw();

}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

   public void draw() {

      System.out.println("Inside Rectangle::draw() method.");

   }

}
```

Square.java

```
public class Square implements Shape {

   public void draw() {

      System.out.println("Inside Square::draw() method.");

   }

}
```

Circle.java

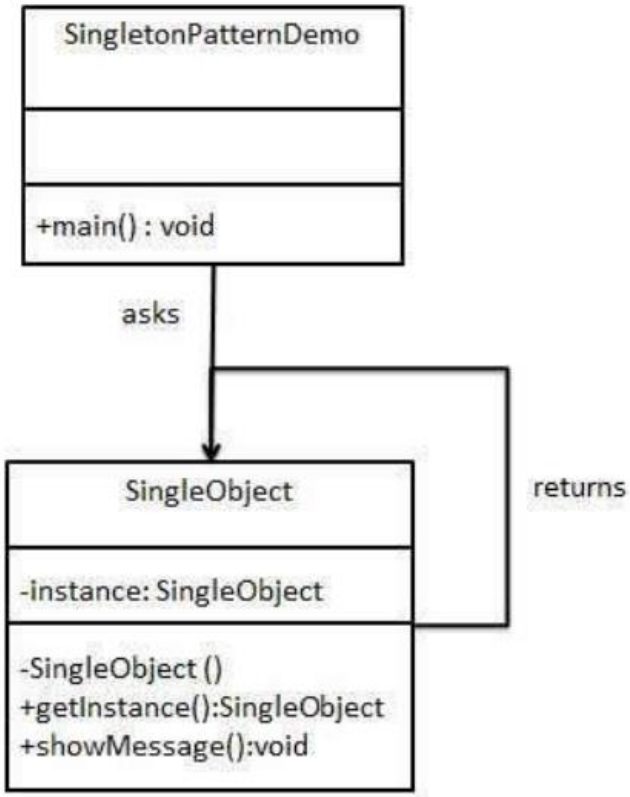
```
public class Circle implements Shape {

   public void draw() {

      System.out.println("Inside Circle::draw() method.");

   }

}
```

Singleton Pattern

- Problem
  - Exactly one instance of a class is allowed. Objects need a global and single point of access
- Solution
  - Define a static model of the class that returns the singleton: ***getInstance()***

Singleton Pattern…

- This pattern involves a single class which is responsible to create an object while making sure that only single object gets created



Step 1

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject

   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be

   //instantiated

   private SingleObject(){}

   //Get the only object available

   public static SingleObject getInstance(){

      return instance;

   }
```

Observer Pattern

- Define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Used in MVC framework
  - Model is problem domain
  - View in windowing system
  - Controller is mouse/keyword control
- Listeners in Java, Thread -> notify()

Adapter Pattern

- Problem
  - How to resolve incompatible interfaces, or how to provide a stable interface to similar components with different interfaces
- Solution
  - Convert the original interface of component into another interface, through an intermediate adapter object
- Example
  - POS needs to adapt several kinds of external third party services, like tax calculators, accounting systems, inventory systems

Adapter Pattern…

- Follows polymorphism, protected variation, indirection

End of Session