Unit-3

GRASP: Designing objects with responsibilities

General Responsibility Assignment Software Patterns(GRASP)

Types of Responsibility

- **Doing:**

Doing responsibility of an object is seen as:

a) doing something itself - create an object, process data, do some computation/calculation

b) initiate and coordinate actions with other objects

- **Knowing:**

Knowing responsibility of an object can be defined as:

a) private and public object data

b) related objects references

c) things it can derive

Different patterns/principles used are

- **Creator -** Problem: Who creates object A?

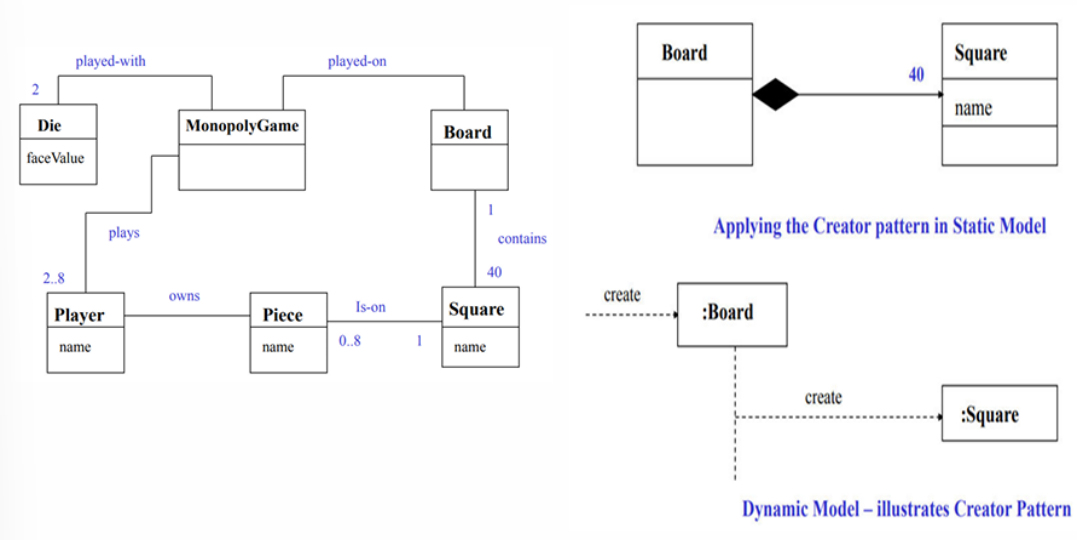
  Solution: Assign class B the responsibility to create object A if one of these is true (more is better):

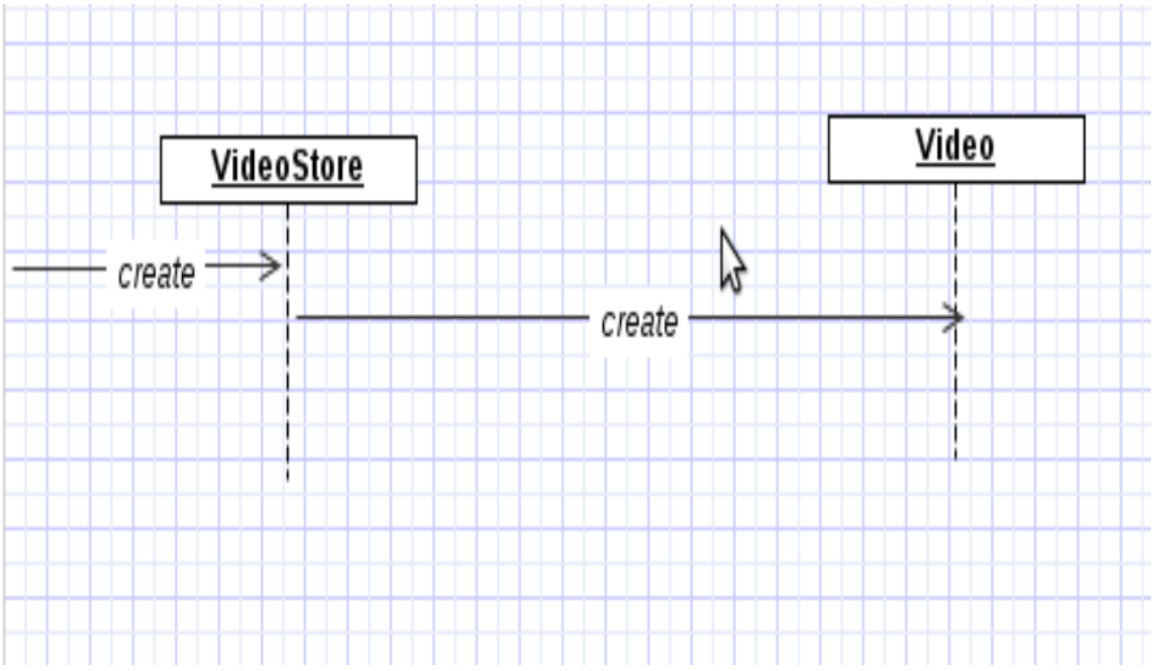
  B contains or compositely aggregates A

  B records A

  B closely uses A

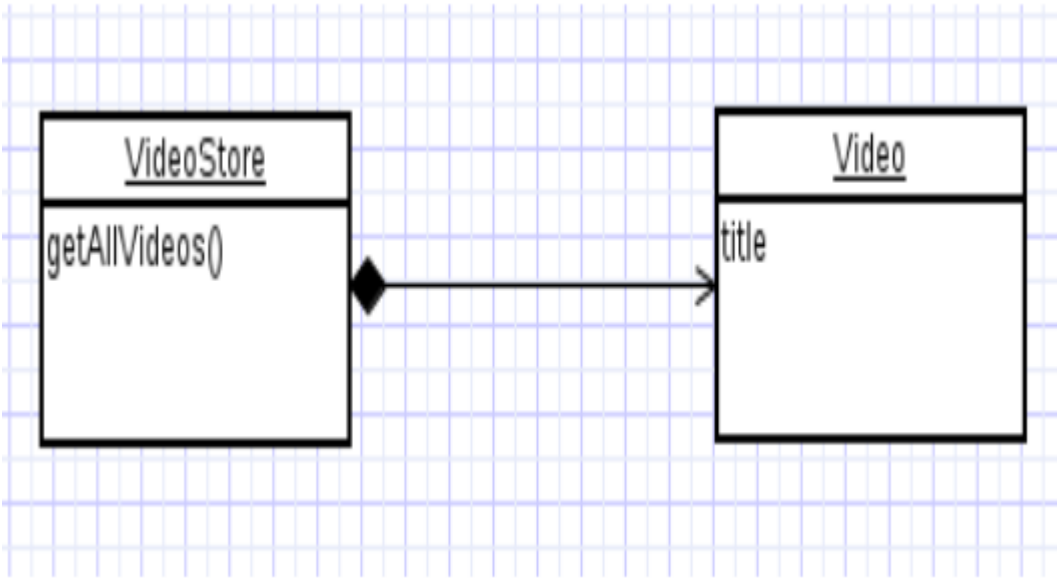
  B has the initializing data for A





- **Information expert -** Problem: What is a basic principle by which to assign responsibilities to objects?

  Solution: Assign a responsibility to the class that has the information needed to fulfill it.



- **Low coupling - Problem:** How can we reduce the impact of change and support low dependency and increased reuse?

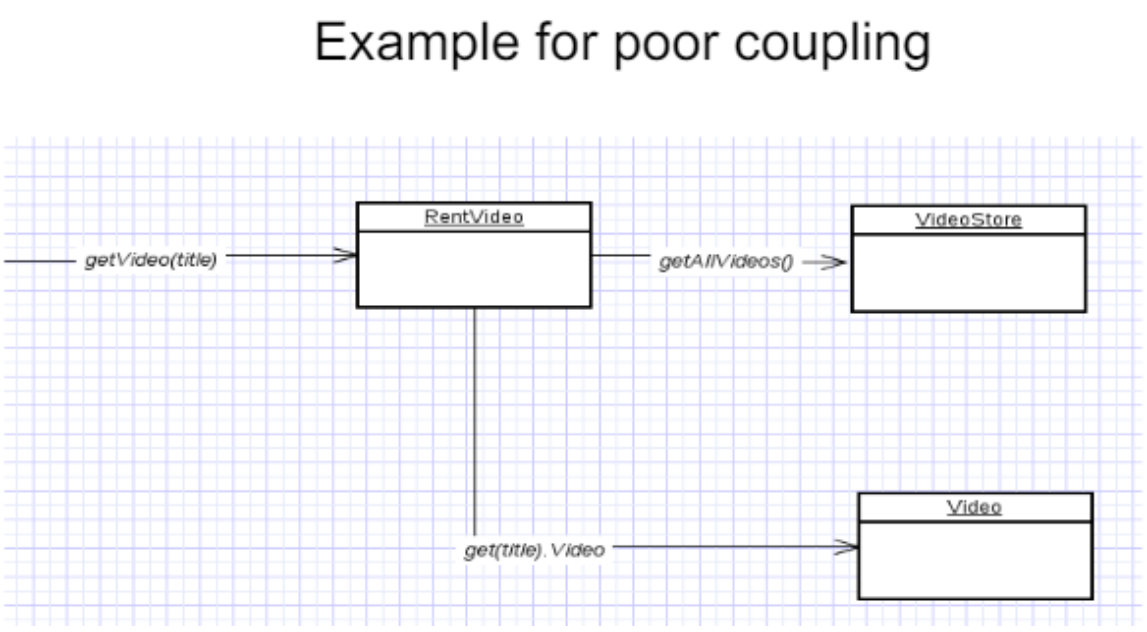
  Solution: To reduce the impact of change and support low dependency and increased reuse, follow these guidelines:

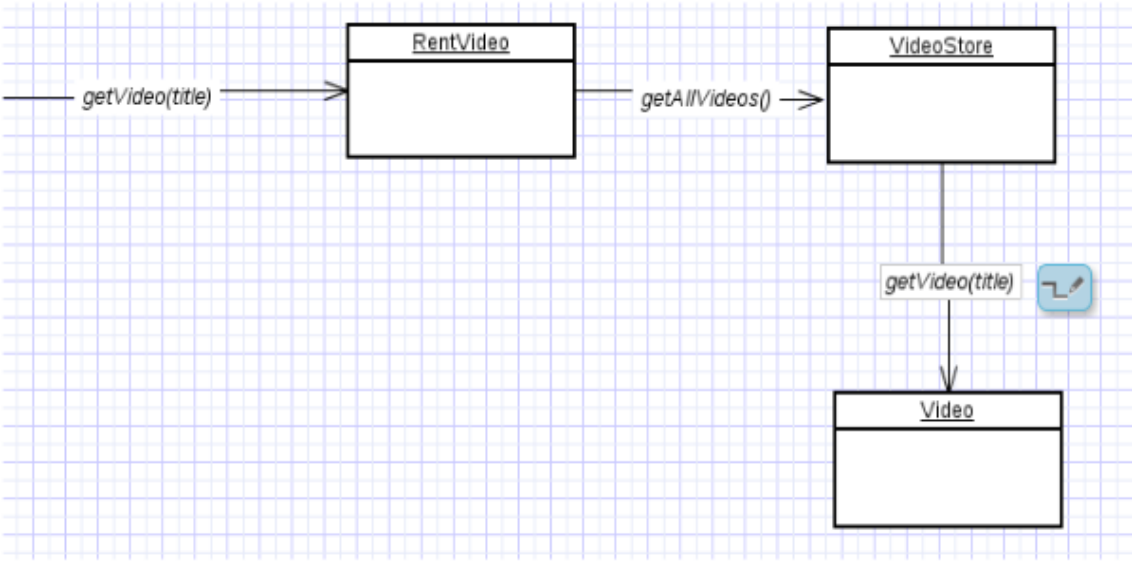
1. Assign responsibilities so that unnecessary coupling remains low.
2. Use this principle to evaluate alternatives.

Coupling is a measure of how one element is related to another. The higher the coupling, the greater the dependence of one element on another.

Low coupling means our objects are more independent and isolated. When something is isolated, we can change it without worrying that we have to change something else or whether we would break something (see Shotgun Surgery).

Using SOLID principles is a great way to keep coupling low. As you can see in the example above, between `CustomerOrdersController` and `AddCustomerOrderCommandHandler`, coupling remains low; they only need to agree on the command object structure.





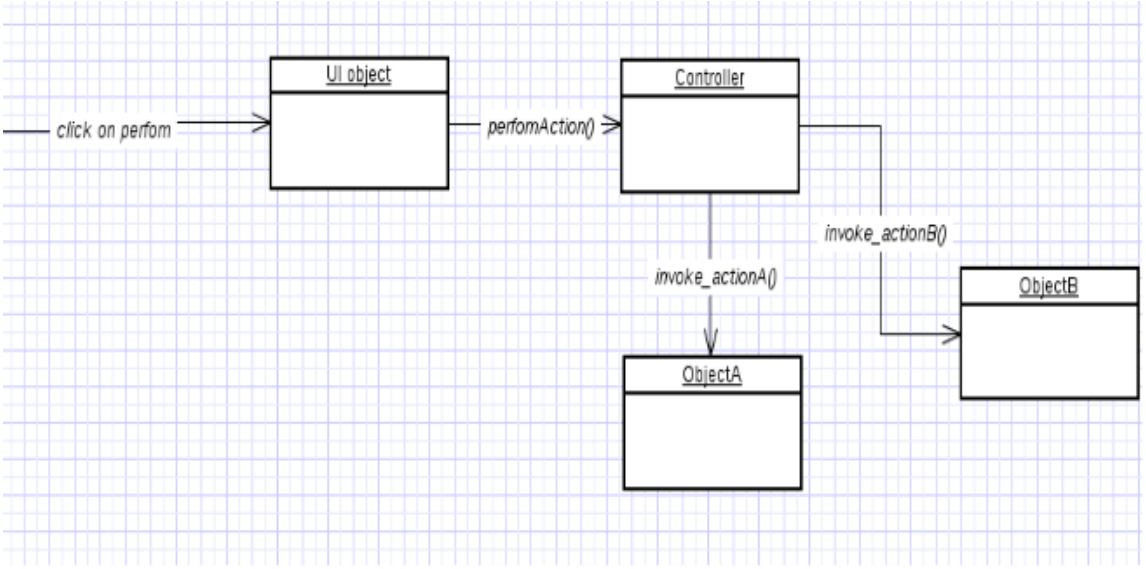
- **Controller - Problem:** What is the first object beyond the UI layer that receives and coordinates the controls of a system operation?

  Solution: Assign this responsibility to an object representing one of these choices:

  1. Represents the overall system, root object, the device the software is running within, or a major subsystem (these are all variations of a facade controller).
  2. Represents a use case scenario within which the system operation occurs (a use case or session controller).

  In general, the implementation of this principle depends on the high-level design of our system. However, we always need to define an object that orchestrates our business transaction processing.

  At first glance, it might seem that the MVC Controller in web applications/APIs is a great example here (even the name is the same), but for me, it is not true. Of course, it receives input, but it shouldn't coordinate a system operation; it should delegate it to a separate service or Command Handler.



1. Façade Controller:

- **Description**:
  - A class that represents the overall system, coordinating interactions between various subsystems.
- **Responsibilities**:
  - Coordinates interactions between different subsystems.
  - Provides a simplified interface to the complex system.
- **Example**:

```
class SystemFacadeController:
    def __init__(self, subsystem1, subsystem2, subsys
tem3):
        self.subsystem1 = subsystem1
        self.subsystem2 = subsystem2
        self.subsystem3 = subsystem3

    def operate_system(self):
        self.subsystem1.perform_operation()
        self.subsystem2.perform_operation()
        self.subsystem3.perform_operation()
```

2. Use Case Controller:

- **Description**:
  - A class that represents a specific use case, handling particular system operations.
- **Responsibilities**:
  - Handles system operations related to a specific use case.
  - Maintains state information required for the use case.
- **Example**:

```
class UseCaseController:
    def __init__(self):
        self.state = {}

    def handle_event1(self):
        # Handle event 1
        pass

    def handle_event2(self):
        # Handle event 2
        pass

    def handle_event3(self):
        # Handle event 3
        pass
```

3. Bloated Controller:

- **Description**:
  - A single class receiving all system events, performing many tasks, and maintaining significant system information.
- **Issues**:
  - Single responsibility principle violation.
  - Low cohesion and high coupling.
- **Example**:

```
class BloatedController:
    def __init__(self):
        self.attribute1 = None
        self.attribute2 = None
        self.attribute3 = None
        # More attributes...

    def handle_event1(self):
        # Perform tasks related to event 1
        pass

    def handle_event2(self):
        # Perform tasks related to event 2
        pass

    def handle_event3(self):
        # Perform tasks related to event 3
        pass
```

In this structure, each controller has a clear responsibility and follows the Single Responsibility Principle.

- **High cohesion - Problem:** How can we keep objects focused, understandable, manageable, and, as a side effect, support Low Coupling?

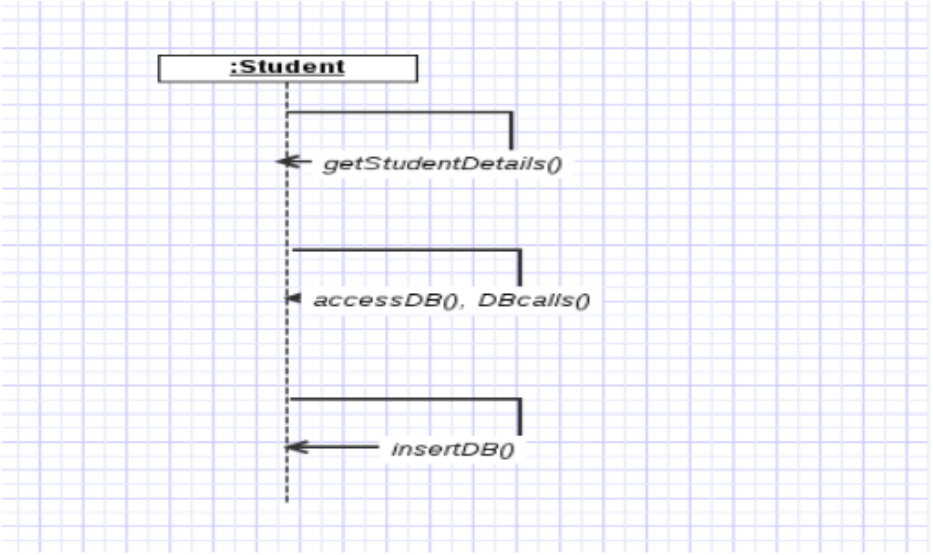
  Solution: To keep objects focused, understandable, manageable, and support Low Coupling, follow these guidelines:

  1. Assign a responsibility so that cohesion remains high.
  2. Use this to evaluate alternatives.

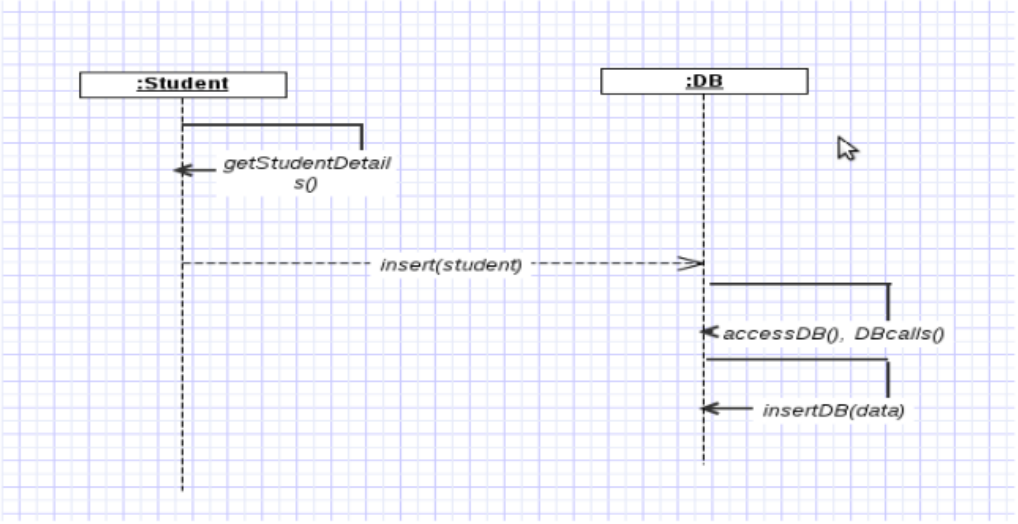
  Cohesion is a measure of how strongly all responsibilities of an element are related. In other words, it measures the degree to which the parts inside an element belong together.

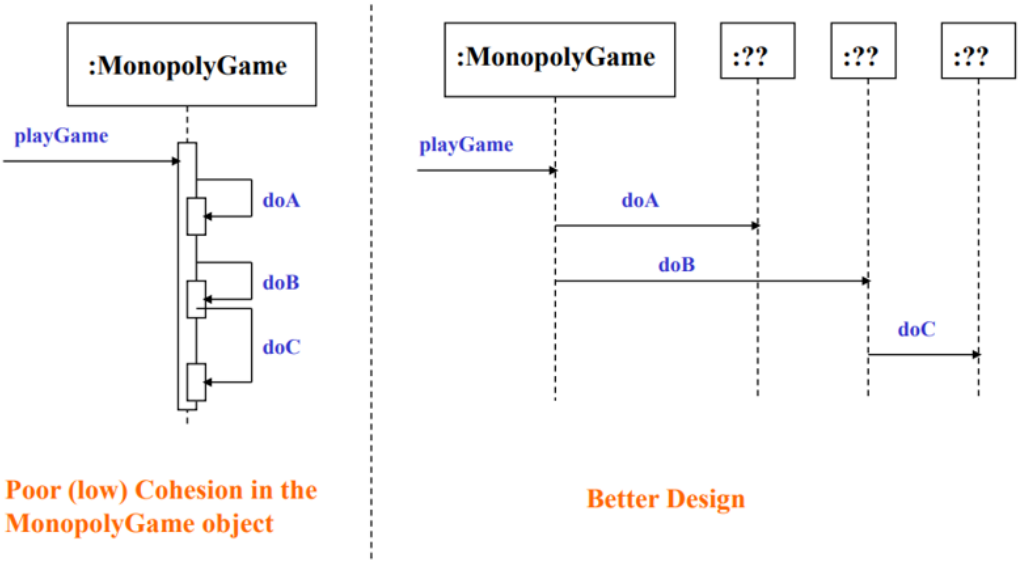
  Classes with low cohesion have unrelated data and/or unrelated behaviors.

Example for low cohesion



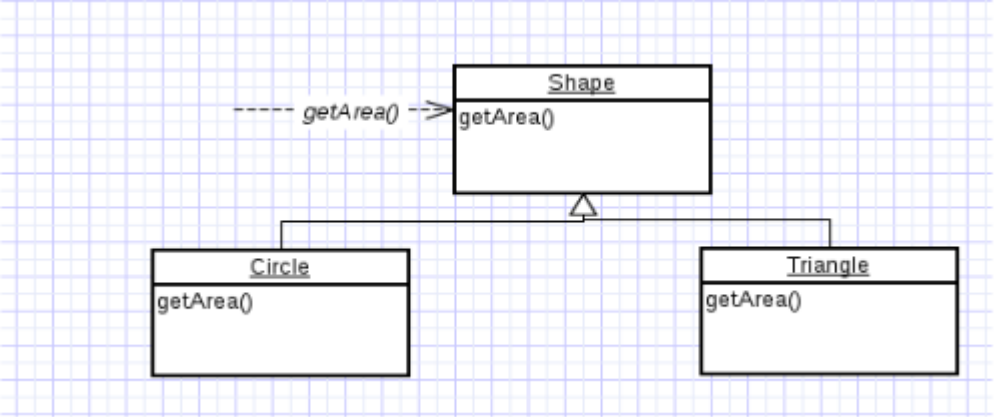
Example for High Cohesion

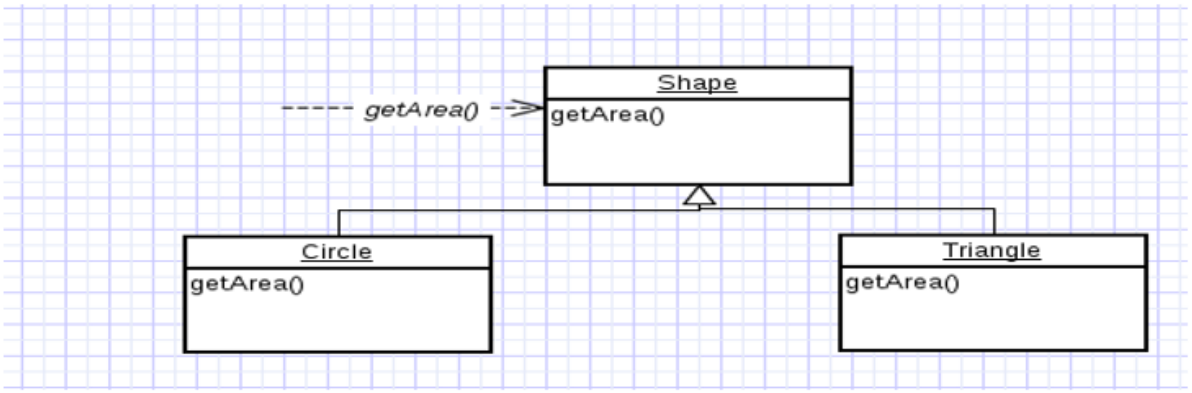




For example, consider the `Customer` class. It has high cohesion because it only manages orders. If I were to add the responsibility of managing product prices to this class, its cohesion would drop significantly because the price list is not directly related to the `Customer` itself.

- **Polymorphism -** Deals with How to act different depending on object's class



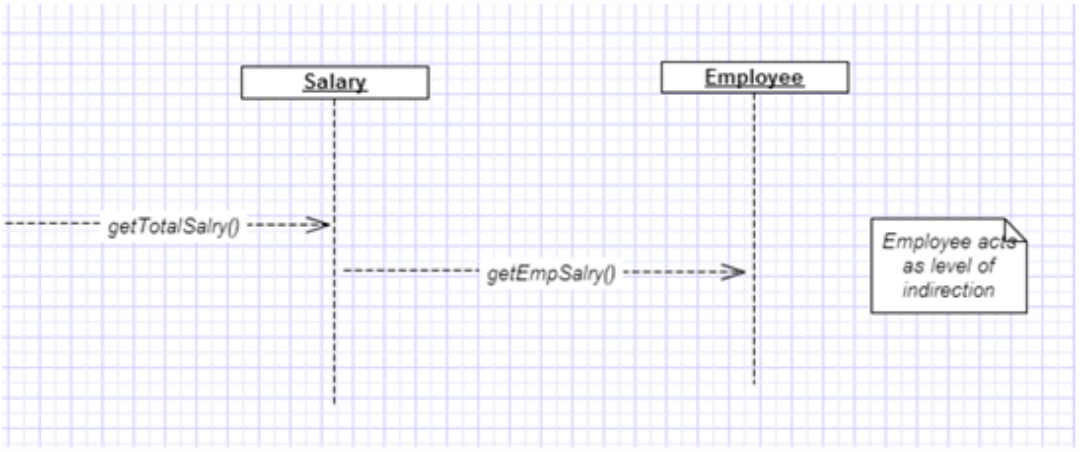


- **Indirection**

  Problem**:** How to assign responsibilities in order to avoid direct coupling between two components and keep ability for reuse.

  Solution: Assign responsibility to intermediate class for providing linking between objects not linking directly

  Related Design patterns: Adapter, Bridge, Mediator



- **Protected variations**

  Problem: How to design objects, subsystems, and systems so that the variations or instability in these elements do 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.

  Variation Point:

  - Branching point in the existing system or in requirements.

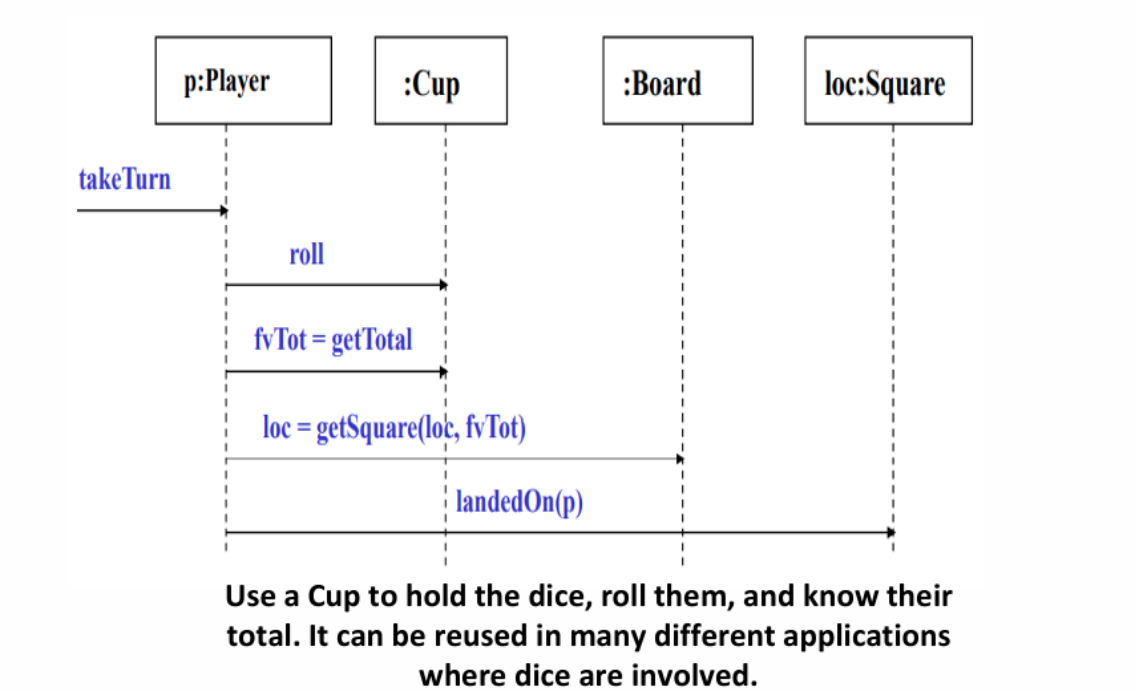
Evolution Point:

- A potential branching point that might occur in the future but is not declared by existing requirements.

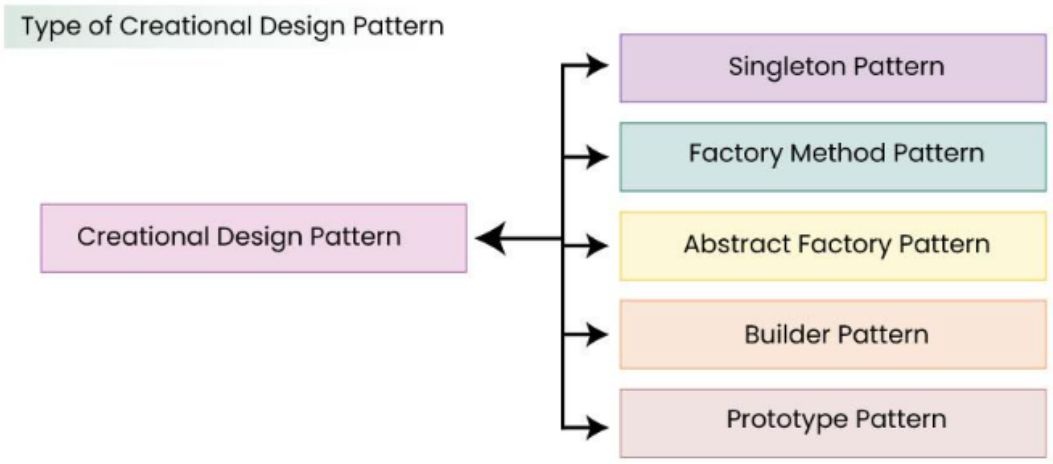
- **Pure fabrication**

  Problem: What object should have responsibility when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate? Sometimes assigning responsibilities only to domain layer software classes leads to problems like poor cohesion or coupling, or low reuse potential.

  Solution: Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a domain concept



Use a Cup to hold the dice, roll them, and know their total. It can be reused in many different applications where dice are involved.



Look at slide 2 for codes

Important Aspects of Bad Design (RFIV)

Rigidity

- the impact of a change is unpredictable
- every change causes a cascade of changes in dependent modules
- costs become unpredictable

Fragility

- the software tends to break in many places on every change
- the breakage occurs in areas with no conceptual relationship
- on every fix the software breaks in unexpected ways

Immobility

- it's almost impossible to reuse interesting parts of the software
- the useful modules have too many dependencies
- the cost of rewriting is less compared to the risk faced to separate those parts

Viscosity

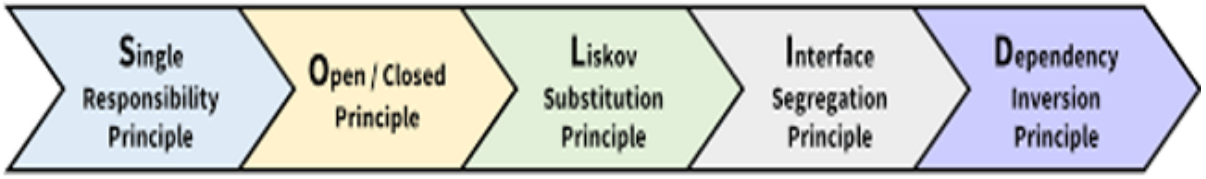
- a hack is cheaper to implement than the solution within the design
- preserving design moves are difficult to think and to implement

- it's much easier to do the wrong thing rather than the right one

For High Cohesiveness and Low Coupling do SOLID

Introduction to SOLID Principles

SOLID is an acronym for five important design principles introduced by Robert J. Martin. These principles lead to more flexible and stable software architecture that's easier to maintain and extend, and less likely to break. The acronym was identified by Michael Feathers.



Key Points:

- **SOLID principles are software design coding standards.**
- **They help to obtain good software with low coupling and high cohesion.**
- **They reduce dependencies - changes in one part of software will not impact others.**
- **Every module or class should have responsibility over a single part of the functionality provided by the software.**

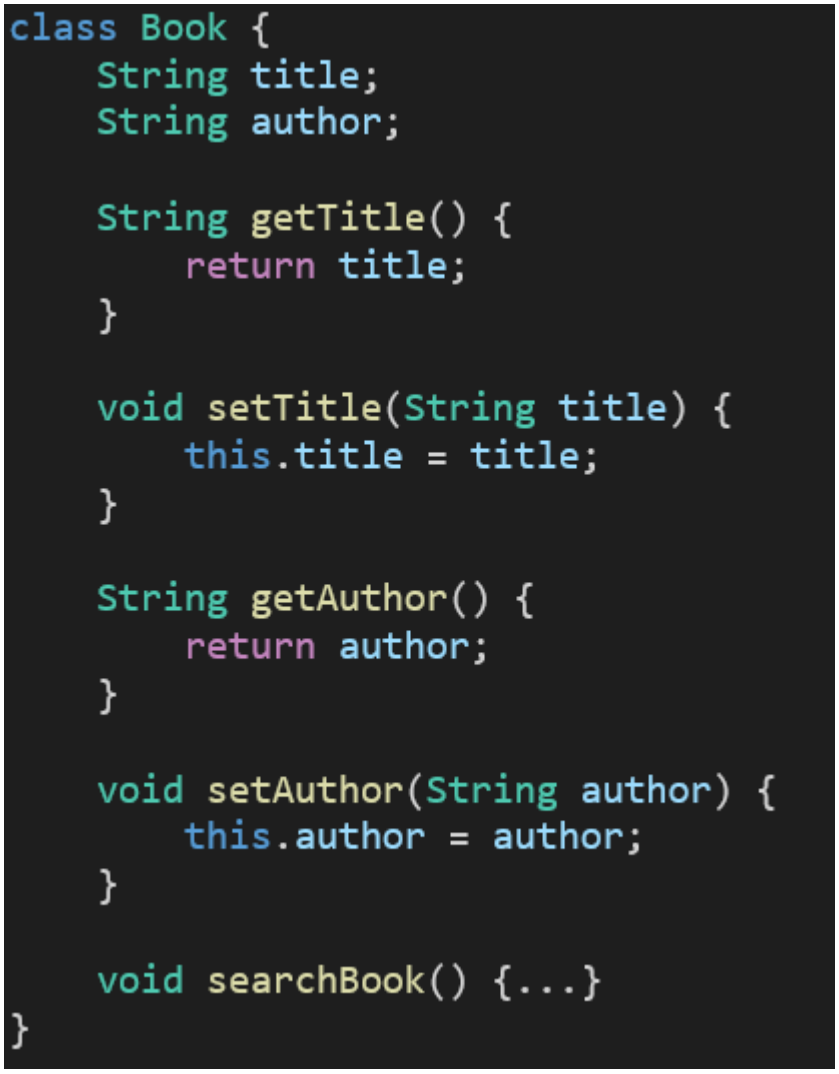
Single Responsibility Principle

- **A class should have one, and only one, reason to change.** (Robert C. Martin)
  - *Each class only does one thing.*
  - *Every class or module only has responsibility for one part of the software's functionality.*
- **Ensures low coupling code.**
- **Ensures easier coding to understand and maintain.**
- **Can be applied to classes, software components, and microservices.**

- **Code becomes easier to test and maintain, it makes software easier to implement, and it helps to avoid unanticipated side-effects of future changes.**

Example of Single Responsibility Principle (SRP) Violation:

- The code violates the Single Responsibility Principle, as the `Book` class has two responsibilities.
  1. It sets the data related to the books (title and author).
  2. It searches for the book in the inventory.



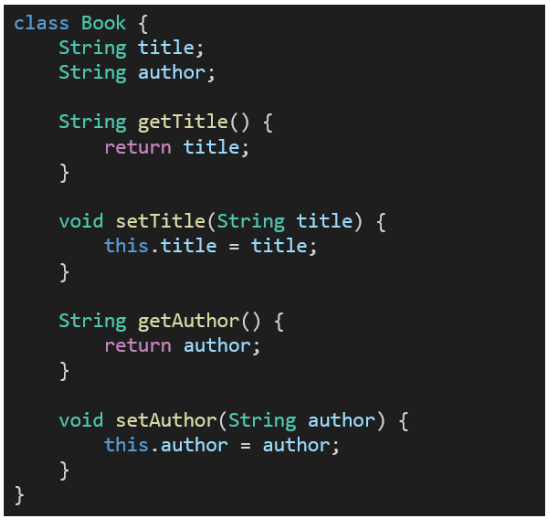
```
class Book {
    String title;
    String author;

    String getTitle() {
        return title;
    }

    void setTitle(String title) {
        this.title = title;
    }

    String getAuthor() {
        return author;
    }

    void setAuthor(String author) {
        this.author = author;
    }

    void searchBook() {...}
}
```

Refactoring to Adhere to SRP:

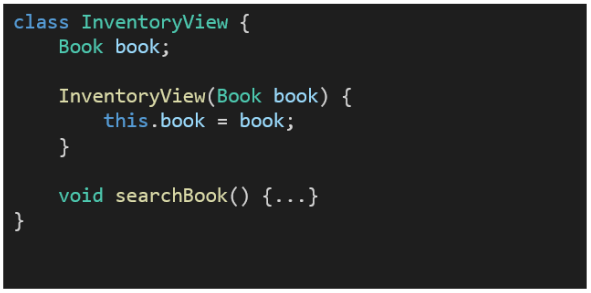
- **Identify things that are changing for different reasons.**

- **Group together things that change for the same reason.**
- **Decouple the responsibilities.**

In the refactored code:



```
class Book {
    String title;
    String author;

    String getTitle() {
        return title;
    }

    void setTitle(String title) {
        this.title = title;
    }

    String getAuthor() {
        return author;
    }

    void setAuthor(String author) {
        this.author = author;
    }
}
```



```
class InventoryView {
    Book book;

    InventoryView(Book book) {
        this.book = book;
    }

    void searchBook() {...}
}
```

- The `Book` class will only be responsible for getting and setting the data of the `Book` object.
- Another class called `InventoryView` will be responsible for checking the inventory.
- The `searchBook()` method will be moved to the new class, and the `Book` class will be referenced in the constructor.

Benefits of Single Responsibility Principle:

- **Testing**: A class with one responsibility will have fewer test cases.
- **Lower Coupling**: Less functionality in a single class will have fewer dependencies.
- **Organization**: Smaller, well-organized classes are easier to search than monolithic ones.

```
class Book {
    String title;
    String author;

    String getTitle() {
        return title;
```

```
    }

    void setTitle(String title) {
        this.title = title;
    }

    String getAuthor() {
        return author;
    }

    void setAuthor(String author) {
        this.author = author;
    }
}

class InventoryView {
    Book book;

    InventoryView(Book book) {
        this.book = book;
    }

    void searchBook() {
        // Search book in inventory
    }
}
```

Open Closed Principle

It’s now time for the O in SOLID, known as the **open-closed principle.** Simply put, **classes should be open for extension but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs** in an otherwise happy application.

- **Bertrand Meyer** originated the term OCP

Refactoring the code to adhere to the Open/Closed Principle (OCP) involves creating an extra layer of abstraction that allows the system to be open for extension but closed for modification.

Let's refactor the code step by step:

1. **Create an Interface for Book Discounts**:

   ```
   public interface BookDiscount {
       String getBookDiscount();
   }
   ```

   This interface defines a contract that all types of book discounts must adhere to. It requires implementing classes to provide a method `getBookDiscount()` which returns a string representing the discount details.

2. **Implement the Interface for Cookbook and Biography Discounts**:

   ```
   public class CookbookDiscount implements BookDiscount {
       @Override
       public String getBookDiscount() {
           String discount = "30% between Dec 1 and 24";
           return discount;
       }
   }

   public class BiographyDiscount implements BookDiscount {
       @Override
       public String getBookDiscount() {
           String discount = "50% on the subject's birthda
   y";
           return discount;
       }
   }
   ```

   Both `CookbookDiscount` and `BiographyDiscount` classes implement the `BookDiscount` interface and provide their own implementation for the `getBookDiscount()` method.

3. **Refactor DiscountManager**:

```
public class DiscountManager {
    void processBookDiscount(BookDiscount discount) {
        String discountDetails = discount.getBookDiscoun
t();
        // Apply discount logic here
    }
}
```

Instead of having separate methods for each type of discount, we now have a single method `processBookDiscount()` that accepts any class that implements the `BookDiscount` interface.

This change allows the `DiscountManager` to be open for extension. Now, if a new type of book discount is introduced, we won't need to modify the `DiscountManager` class. We only need to create a new class that implements the `BookDiscount` interface.

Previous Code:

```
class CookbookDiscount {
    String getCookbookDiscount() {
        String discount = "30% between Dec 1 and 24";
        return discount;
    }
}

class DiscountManager {
    void processCookbookDiscount(CookbookDiscount discount)
{...}
}

class BiographyDiscount {
    String getBiographyDiscount() {
        String discount = "50% on the subject's birthday.";
        return discount;
    }
}
```

```
class DiscountManager {
    void processCookbookDiscount(CookbookDiscount discount)
{...}
    void processBiographyDiscount(BiographyDiscount discoun
t) {...}
}
```

Refactored Code:

```
public interface BookDiscount {
    String getBookDiscount();
}

public class CookbookDiscount implements BookDiscount {
    @Override
    public String getBookDiscount() {
        String discount = "30% between Dec 1 and 24";
        return discount;
    }
}

public class BiographyDiscount implements BookDiscount {
    @Override
    public String getBookDiscount() {
        String discount = "50% on the subject's birthday";
        return discount;
    }
}

public class DiscountManager {
    void processBookDiscount(BookDiscount discount) {
        String discountDetails = discount.getBookDiscount
();
        // Apply discount logic here
    }
}
```

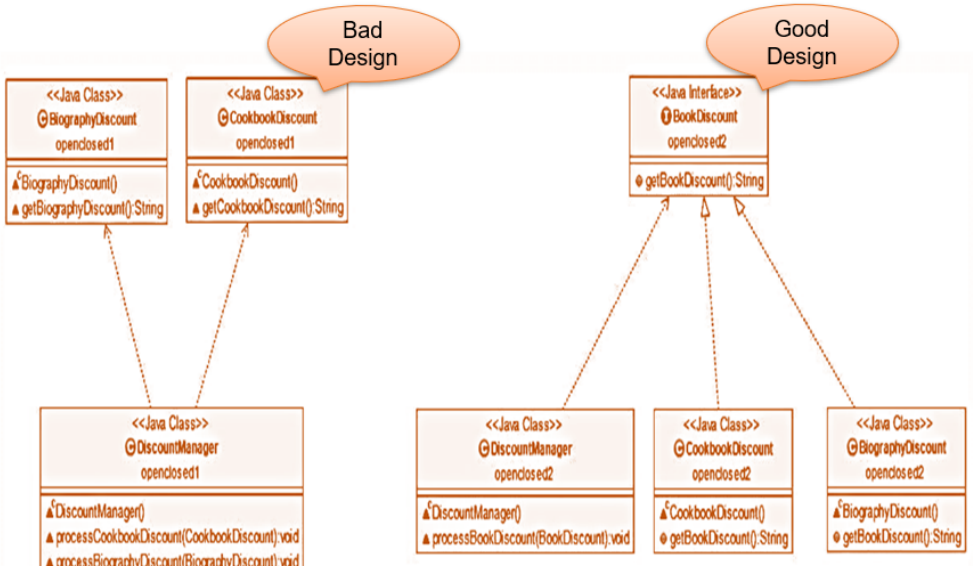
Comparison:

Previous Code:

- Separate classes for each type of discount.
- DiscountManager has specific methods for each type of discount.
- Violates the Open/Closed Principle because modifying the DiscountManager class is necessary when adding a new type of discount.

Refactored Code:

- Introduces an interface `BookDiscount` to represent all types of book discounts.
- `CookbookDiscount` and `BiographyDiscount` implement the `BookDiscount` interface.
- `DiscountManager` now has a single method `processBookDiscount()` that accepts any class implementing the `BookDiscount` interface.
- Adheres to the Open/Closed Principle; the `DiscountManager` class is open for extension but closed for modification.



Absolutely, not following the Open/Closed Principle can lead to several issues:

1. **Increased Testing Effort:**
    - When new functionality is added by modifying existing code, it necessitates re-testing the entire system to ensure that the changes haven't introduced any unexpected behaviors or bugs.
2. **Increased Cost and Time:**

- Having to test the entire system repeatedly adds to the cost and time of the development process.

3. **Violation of Single Responsibility Principle (SRP):**
    - Modifying existing classes to accommodate new functionality can lead to violating the SRP as those classes may now have multiple reasons to change.
4. **Maintenance Overhead:**
    - With every modification, the complexity of the existing codebase increases, making it harder to maintain and understand.

By adhering to the Open/Closed Principle and other SOLID principles, such as the Single Responsibility Principle, you can keep your codebase more maintainable, extensible, and easier to test, thus reducing the overall cost and time of development.

Liskov Substitution

Next on our list is Liskov substitution, which is arguably the most complex of the five principles. Simply put, **if class *A* is a subtype of class *B*, we should be able to replace *B* with *A* without disrupting the behavior of our program.** Derived types must be completely substitutable for their base types

The Liskov Substitution Principle (LSP), introduced by Barbara Liskov in 1987, states that derived types must be completely substitutable for their base types. In other words, an object of a superclass should be replaceable by objects of its subclasses without causing issues in the application. This principle ensures that a child class should never change the characteristics of its parent class, and derived classes should never do less than their base class.

LSP is applicable to inheritance (is-a relationship) hierarchies and helps avoid overuse or misuse of inheritance.

Liskov Substitution Principle (LSP) states that objects of a superclass should be replaceable with objects of its subclasses without affecting the behavior of the program. Here's an example code that violates LSP and its refactored version:

Violation of Liskov Substitution Principle:

```
// Base class
class BookDelivery {
    String title;
    int userID;

    // Method to get delivery locations
    void getDeliveryLocations() {
        // Implementation to get delivery locations
    }
}

// Subclass extending BookDelivery
class HardcoverDelivery extends BookDelivery {
    // Override getDeliveryLocations() method
    @Override
    void getDeliveryLocations() {
        // New implementation for HardcoverDelivery
    }
}

// Subclass extending BookDelivery
class AudiobookDelivery extends BookDelivery {
    // Override getDeliveryLocations() method
    @Override
    void getDeliveryLocations() {
        // Audiobooks can’t be delivered to physical locati
ons.
        // This implementation violates LSP
    }
}
```

Refactored Solution:

```
// Base class
class BookDelivery {
    String title;
    int userID;
```

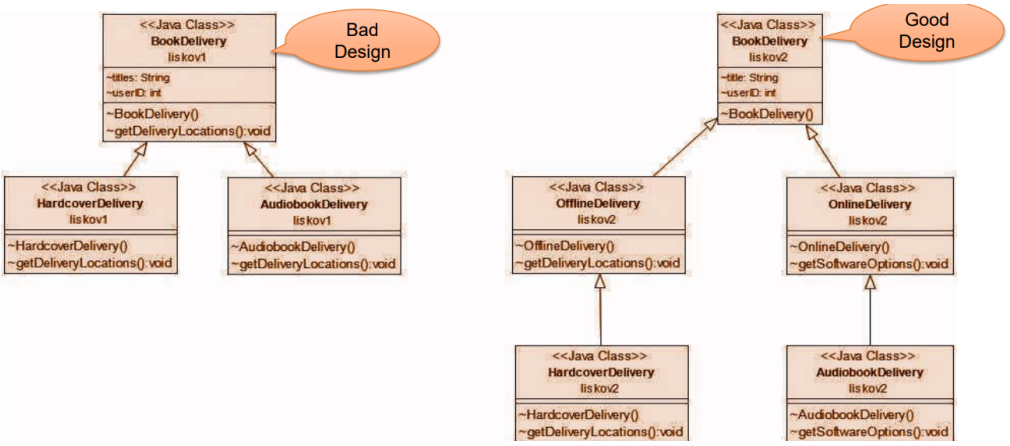
```
}

// Subclass for offline delivery
class OfflineDelivery extends BookDelivery {
    // Method to get delivery locations
    void getDeliveryLocations() {
        // Implementation to get delivery locations
    }
}

// Subclass for online delivery
class OnlineDelivery extends BookDelivery {
    // Method to get software options
    void getSoftwareOptions() {
        // Implementation to get software options
    }
}

// Subclass for delivering hardcover books
class HardcoverDelivery extends OfflineDelivery {
    // Override getDeliveryLocations() method
    @Override
    void getDeliveryLocations() {
        // New implementation for HardcoverDelivery
    }
}

// Subclass for delivering audiobooks
class AudiobookDelivery extends OnlineDelivery {
    // Override getSoftwareOptions() method
    @Override
    void getSoftwareOptions() {
        // Implementation specific to audiobook delivery
    }
}
```

Explanation:

In the refactored solution, the inheritance hierarchy is fixed by introducing an extra layer of abstraction. `OfflineDelivery` and `OnlineDelivery` classes are introduced to better differentiate between delivery types. This separates the concerns of physical and online delivery.

`HardcoverDelivery` is now a subclass of `OfflineDelivery`, and it overrides the `getDeliveryLocations()` method with its own functionality.

Similarly, `AudiobookDelivery` is a subclass of `OnlineDelivery` and overrides the `getSoftwareOptions()` method with its own implementation. This separates the concerns, and each subclass now adheres to Liskov Substitution Principle.



Interface Segregation Principle (ISP)

Larger interfaces should be split into smaller ones. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

The Interface Segregation Principle (ISP) suggests that many client-specific interfaces are better than one general-purpose interface. It aims to prevent clients from being forced to implement functions they don't need. Let's see an example:

```
public interface ParkingLot {
    void parkCar(); // Decrease empty spot count by 1
    void unparkCar(); // Increase empty spots by 1
    void getCapacity(); // Returns car capacity
    double calculateFee(Car car); // Returns the price base
```

```
d on the number of hours
    void doPayment(Car car);
}

class Car {
}
```

We have a simplified parking lot interface that includes both parking and payment-related methods. Now, consider that we want to implement a parking lot where parking is free:

```
public class FreeParking implements ParkingLot {

    @Override
    public void parkCar() {
        // Implementation for parking a car
    }

    @Override
    public void unparkCar() {
        // Implementation for unparking a car
    }

    @Override
    public void getCapacity() {
        // Implementation for getting the capacity
    }

    @Override
    public double calculateFee(Car car) {
        return 0; // Parking is free, so the fee is always
0
    }

    @Override
    public void doPayment(Car car) {
        throw new Exception("Parking lot is free"); // Paym
ent not needed for free parking
```

```
    }
}
```

Our original `ParkingLot` interface included both parking-related and payment-related methods, making it too specific. As a result, our `FreeParking` class is forced to implement payment-related methods that are irrelevant.

Let's segregate the interfaces:

```
// Parking-related interface
public interface ParkingLot {
    void parkCar();
    void unparkCar();
    void getCapacity();
    double calculateFee(Car car);
}

// Payment-related interface
public interface Payment {
    void doPayment(Car car);
}

class Car {
}

// Implementation of FreeParking with only parking-related
methods
public class FreeParking implements ParkingLot {

    @Override
    public void parkCar() {
        // Implementation for parking a car
    }

    @Override
    public void unparkCar() {
        // Implementation for unparking a car
    }
```

```
    @Override
    public void getCapacity() {
        // Implementation for getting the capacity
    }

    @Override
    public double calculateFee(Car car) {
        return 0; // Parking is free, so the fee is always
0
    }
}
```

Now, with segregated interfaces, we have separated parking-related functionality from payment-related functionality. This makes our model more flexible and extendable, and clients do not need to implement any irrelevant logic.

Dependency Inversion Principle (DIP)

1. **Program to the interface, not the implementation**:
    - High-level modules should not depend on low-level modules, both should depend upon abstractions.
    - Abstractions should not depend upon details; details should depend upon abstractions.
2. **Avoid tightly coupled code**:
    - The goal is to avoid tightly coupled code, as it easily breaks the application.
3. **Decouple high-level and low-level classes**:
    - The interaction between them must be thought of as an abstract interaction.
4. **Combination of Open-Closed and Liskov Substitution Principle**:
    - Dependency Inversion Principle (DIP) is a specific combination of the Open-Closed Principle (OCP) and Liskov Substitution Principle (LSP).
    - **Open-Closed Principle (OCP)**: Software entities (classes, modules, functions, etc.) should be open for extension, but closed for

modification.

- **Liskov Substitution Principle (LSP)**: Subtypes must be substitutable for their base types without altering the correctness of the program.

The Dependency Inversion Principle (DIP) suggests programming to the interface, not the implementation. It states that high-level modules should not depend on low-level modules; both should depend on abstractions. Here's an example demonstrating the Dependency Inversion Principle:

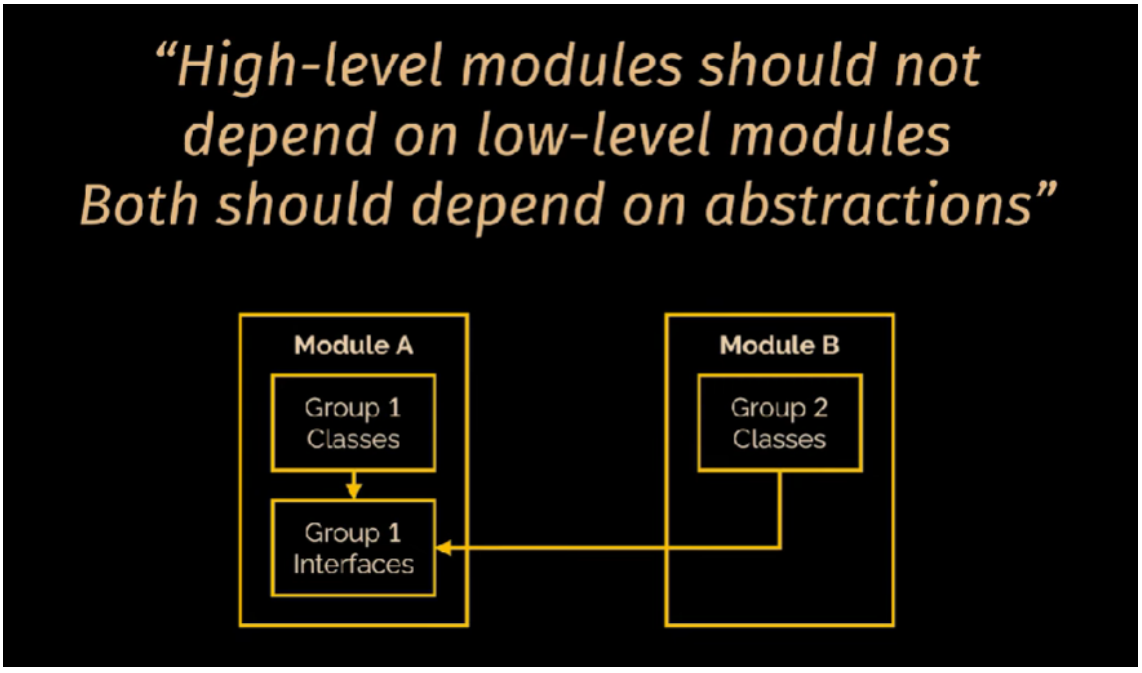
```
// Interface representing a Keyboard
public interface Keyboard {}

// StandardKeyboard class implementing the Keyboard interfa
ce
public class StandardKeyboard implements Keyboard {}

// Interface representing a Monitor
public interface Monitor {}

// StandardMonitor class implementing the Monitor interface
public class StandardMonitor implements Monitor {}

// Windows98Machine class following Dependency Inversion Pr
inciple
public class Windows98Machine {

    private final Keyboard keyboard;
    private final Monitor monitor;

    // Constructor accepting dependencies through Dependenc
y Injection
    public Windows98Machine(Keyboard keyboard, Monitor moni
tor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

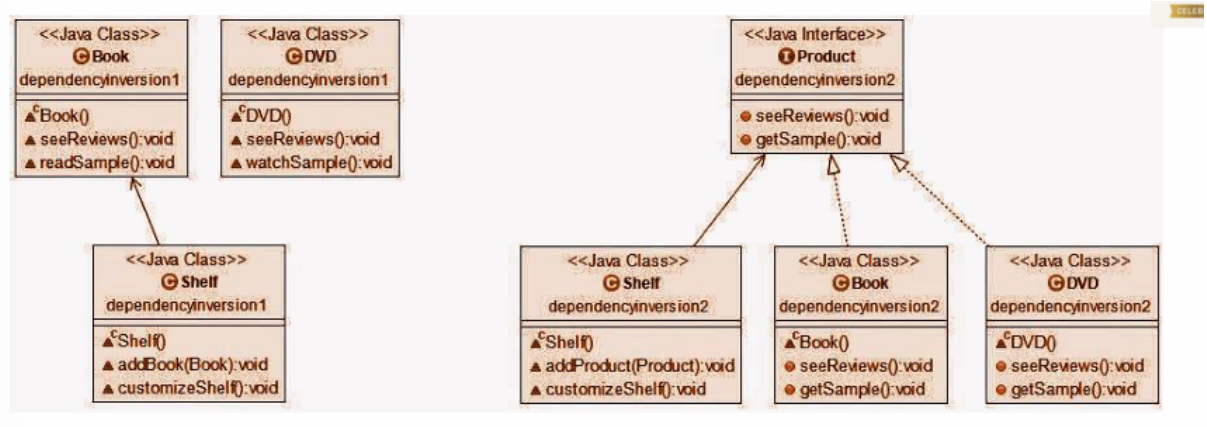
In this code:

- We have `Keyboard` and `Monitor` interfaces.
- We then implemented these interfaces with `StandardKeyboard` and `StandardMonitor` classes respectively.
- We modified the `Windows98Machine` class constructor to accept `Keyboard` and `Monitor` objects as parameters. This way, we're using the Dependency Injection pattern to inject dependencies into the `Windows98Machine` class.

By doing this, we've decoupled our classes and allowed them to communicate through abstractions. Now, if we want to switch out the type of keyboard or monitor, we can easily do so by providing different implementations of the `Keyboard` and `Monitor` interfaces.

This approach makes our code more flexible, easier to maintain, and facilitates easier testing. It also adheres to the Dependency Inversion Principle by ensuring that high-level modules depend on abstractions rather than concrete implementations.




<<Java Class>>
Book
dependencyinversion1
Book()
seeReviews():void
readSample():void
<<Java Class>>
DVD
dependencyinversion1
DVD()
seeReviews():void
watchSample():void
<<Java Class>>
Shelf
dependencyinversion1
Shelf()
addBook(Book):void
customizeShelf():void
<<Java Interface>>
Product
dependencyinversion2
seeReviews():void
getSample():void
<<Java Class>>
Shelf
dependencyinversion2
Shelf()
addProduct(Product):void
customizeShelf():void
<<Java Class>>
Book
dependencyinversion2
Book()
seeReviews():void
getSample():void
<<Java Class>>
DVD
dependencyinversion2
DVD()
seeReviews():void
getSample():void

Pattern Name	Need For the Pattern	Avoids	How to Implement	Scenario
Creator	Assigns responsibility for object creation	Redundancy, Complexity	Constructor, Factory Method	Creating instances of classes (e.g., creating a new car)
Information Expert	Assigns responsibility for information management	Tight Coupling	Encapsulation, Delegation	Calculating total price in a shopping cart
Low Coupling	Reduces dependency between classes	Tight Coupling	Dependency Injection, Interfaces	Integrating different modules in a large system
High Cohesion	Groups related responsibilities together	Low Cohesion	Encapsulation, Separation of Concerns	Managing user data in a user management system
Controller	Mediates between the user interface and domain logic	Direct manipulation	Service Classes, Facade	Handling user requests in a web application
Pure Fabrication	Introduces a new class to fulfill a responsibility	Unnecessary complexity	Helper Classes, Factories	Implementing a utility class for string manipulation
Indirection	Decouples classes by introducing intermediaries	Tight Coupling	Interfaces, Dependency Injection	Creating objects indirectly to improve flexibility
Polymorphism	Enables objects to be treated uniformly	Direct method invocation	Inheritance, Interfaces	Implementing shape classes with common behaviour
Protected Variation	Shields client code from changes in implementation	Dependency on specifics	Abstraction, Dependency Inversion, Interfaces	Implementing payment gateways with varying implementations

Above is Grasp Principle

Principle Name	Need For the Principle	Avoids What	How to Implement	Scenario
Single Responsibility	Ensures that each class or module has only one responsibility or reason to change.	Complexity, Code Smell, Unmaintainable Code	Divide functionality into smaller, cohesive classes	In an online store application, a `Product` class should only be responsible for managing product details, not for handling user authentication.
Open/Closed	Encourages classes to be open for extension but closed for modification.	Modification of existing code, Fragile codebase	Use abstract classes, interfaces, and inheritance	In a banking application, adding a new type of account should not require modifying existing account classes.
Liskov Substitution	Subtypes should be substitutable for their base types without altering the correctness of the program.	Violation of behavioural contracts, Unexpected behavior	Use inheritance and adhere to the "is-a" relationship	A method accepting a `List` should work correctly with any subtype of `List`, such as `ArrayList` or `LinkedList`.
Interface Segregation	Clients should not be forced to depend on interfaces they do not use.	Dependency on unnecessary methods, Tight coupling, Interface bloat	Define specific interfaces for specific clients	In a notification system, a `SMSNotifier` should not be forced to implement methods related to email notifications.
Dependency Inversion	High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.	Tight coupling, Fragile codebase, Difficulty in testing and maintenance	Use dependency injection, abstractions, and IoC	In a payment processing system, a `PaymentProcessor` class should not directly instantiate a `CreditCardProcessor`, but should depend on an interface.

Above is Solid Principle

Design Patterns

Design patterns are used to represent some of the best practices adapted by experienced object-oriented software developers

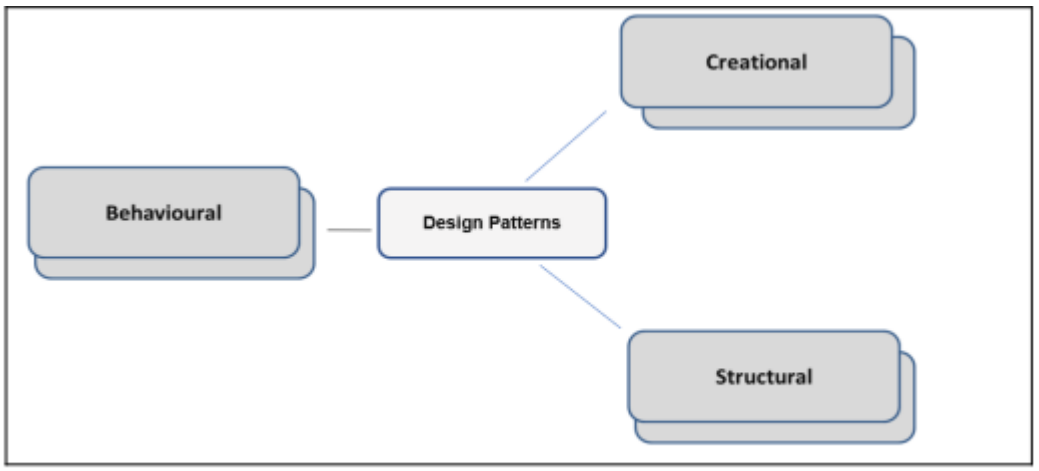


Fig 1: Categories of Design Patterns

Design patterns are primarily based on the following principles of object oriented design.

1. **Program to an interface, not an implementation**: This principle suggests that code should be written in terms of interfaces or abstract classes rather than specific implementations. This allows for greater flexibility and easier maintenance as implementations can be swapped out without affecting the rest of the codebase.
2. **Favor object composition over inheritance**: This principle advises developers to prefer composing objects together to achieve a desired functionality rather than using inheritance. Object composition provides greater flexibility and reusability, and it avoids some of the limitations and complexities associated with inheritance.
3. **Common platform for developers**: Design patterns provide a common language and set of solutions for developers. By using design patterns, developers can communicate more effectively about the structure of their code. For example, when a developer sees that a program is following a singleton pattern, they know that it is designed to use a single instance of a class. This standardization facilitates collaboration and makes it easier for developers to understand and work on each other's code.
4. **Best practices**: Design patterns have been developed and refined over time to provide solutions to common problems in software development. By learning and using these patterns, developers can benefit from the collective wisdom of the software development community. Design patterns help inexperienced developers learn software design more easily and quickly by providing proven solutions to common problems. This allows

developers to focus on solving new problems rather than reinventing the wheel.

The Gang of Four (GoF) defines a design pattern by the following essential elements:

1. **Name**: The pattern's name serves as a handle to describe the problem, its solutions, and its consequences in a word or two.
    - **Describes the pattern**: This is a brief description of the pattern, outlining its purpose and functionality.
    - **Adds to common terminology for facilitating communication**: Design patterns establish a common vocabulary that helps developers communicate more efficiently about software design.
2. **Problem**: Describes the problem that the pattern addresses. It explains when to apply the pattern by defining the context and the specific design issues.
    - **Describes when to apply the pattern**: Specifies the conditions under which the pattern is applicable.
    - **Answers - What is the pattern trying to solve?**: Clearly states the problem the pattern addresses and what it is trying to solve.
3. **Solution**: Describes the elements, relationships, responsibilities, and collaborations that make up the design. It explains how to implement the pattern.
4. **Consequences**:
    - **Results of applying the pattern**: Discusses the benefits and drawbacks of using the pattern.
    - **Benefits and Costs**: Provides an overview of the advantages and disadvantages of applying the pattern. These can vary depending on specific scenarios.

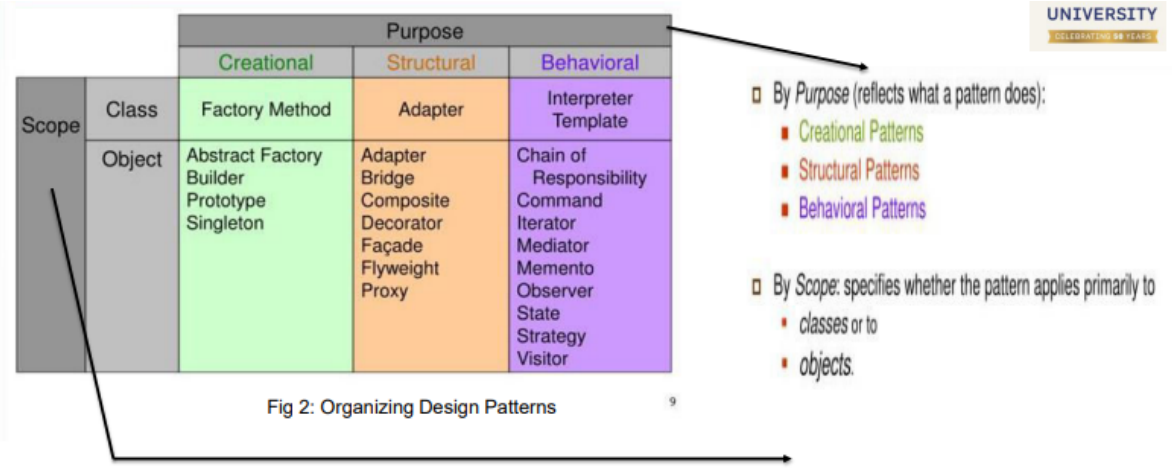


Fig 2: Organizing Design Patterns

Selection of a Design Pattern

1. Consider how design patterns solve design problems
2. Scan intent sections
3. Study how patterns interrelate
4. Study patterns of like purpose
5. Examine a cause of redesign
6. Consider what should be variable in your design

Creational patterns deal with the process of object creation

Structural patterns, deal primarily with the static composition and structure of classes and objects

Behavioral patterns, which deal primarily with dynamic interaction among classes and objects

Architecture vs Design

Architecture: High level framework for structuring an application

- client-server based on rpc
- abstraction layering
- distrubuted object-oriented system based on COBRA

Design: Reusable collaborations that solves subproblems within an application

- Object-oriented high level of abstraction

- Framework that guides object-oriented implementation

What are Design Patterns?

- Reusable solutions to the problems
- Interaction between the objects
- Template, not a solution
- Language independent

Pros of Design Patterns

- Add consistency to designs by solving similar problems the same way, independent of language
- Add clarity to design and design communication by enabling a common vocabulary
- Improve time to solution by providing templates which serve as foundations for good design
- Improve reuse through composition

Cons of Design Patterns

- Some patterns come with negative consequences (i.e. object proliferation, performance hits, additional layers)
- Consequences are subjective depending on concrete scenarios
- Patterns are subject to different interpretations, misinterpretations, and philosophies
- Patterns can be overused and abused - Anti-Patterns

Creational Pattern

- Creational design patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.
- Encapsulate knowledge about which concrete classes the system uses
- Hide how instances of these classes are created and put together

Problem with *new* key word in Java

The new operator is often considered harmful as it scatters objects all over the application. Over time it can become challenging to change an implementation because classes become tightly coupled.

Types of Creational Pattern

1. Singleton
2. Factory Method
3. Builder
4. Prototype

Singleton

The singleton pattern is a design pattern that restricts the instantiation of a class to one object.

The Singleton is part of the Creational Design Pattern Family. Singleton Design Pattern aims to keep a check on initialization of objects of a particular class by ensuring that only one instance of the object exists throughout the Java Virtual Machine.

Why implement Singleton Design Pattern?
1. We can be sure that a class has only a single instance.
2. We gain a global access point to that instance.
3. The Singleton Object is initialized only when it's requested for the first time.

The Singleton pattern solves two main problems:

1. **Ensure that a class has just a single instance**: This is useful when you want to control access to a shared resource, such as a database connection or a file. With the Singleton pattern, you can ensure that only one instance of the class is created, and subsequent requests for instances return the same instance.
2. **Provide a global access point to that instance**: The Singleton pattern provides a global access point to the single instance it creates, allowing any part of the program to access that instance. This ensures that the instance is easily accessible without the risk of being overwritten by other parts of the code.

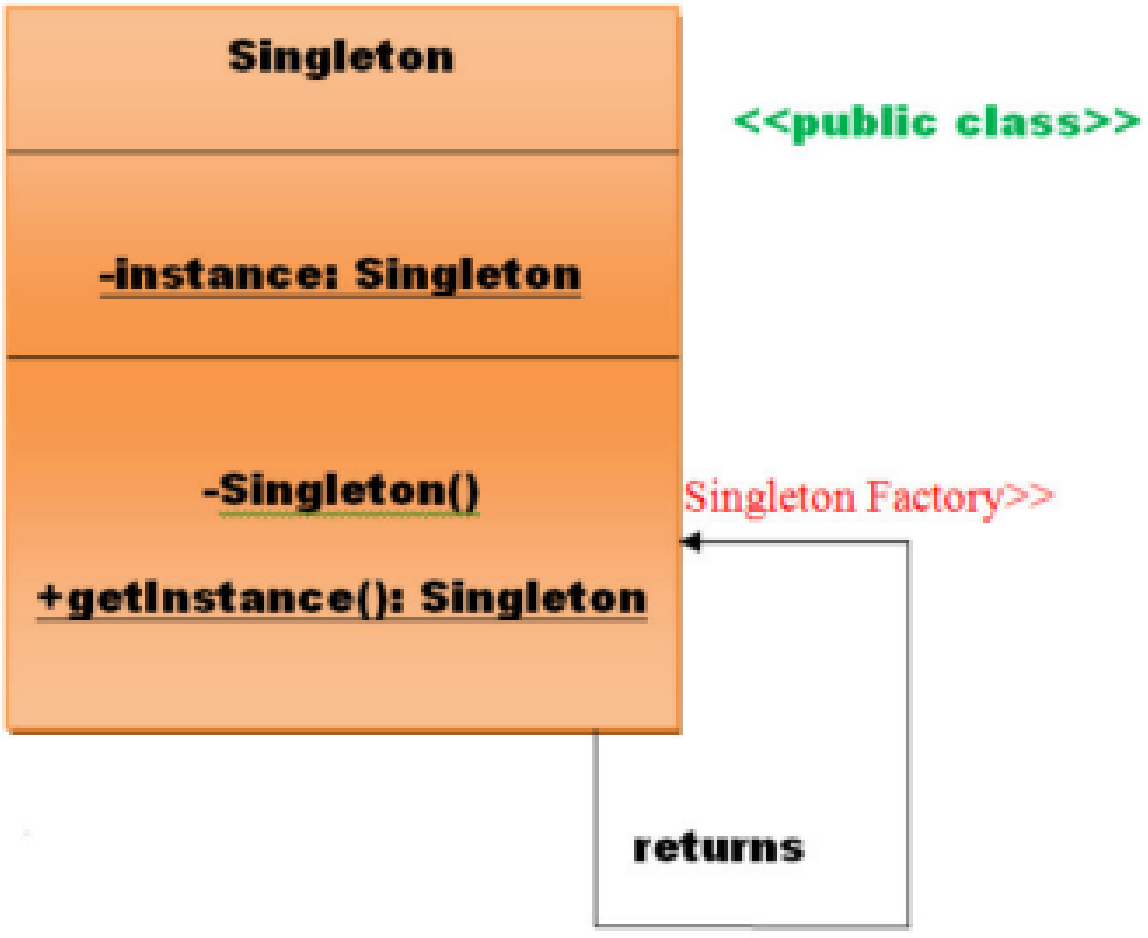
The Singleton pattern helps in centralizing the code responsible for creating and managing the single instance, rather than scattering it throughout the

program.

The solution involves two steps common to all implementations of the Singleton pattern:

1. **Make the default constructor private**: This prevents other objects from using the `new` operator with the Singleton class, ensuring that no additional instances can be created.
2. **Create a static creation method**: This method acts as a constructor by creating an instance of the Singleton class and saving it in a static field. Subsequent calls to this method return the cached object, ensuring that the same instance is always returned.

By following these steps, the Singleton pattern guarantees that only one instance of the class is created and provides a global access point to that instance, while also encapsulating the creation and management of the instance within the class itself.



- **Singleton pattern is used for logging, drivers objects, caching, and thread pool.**
- **Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade, etc. Singleton design pattern is used in core Java classes also (for example, java.lang.Runtime, java.awt.Desktop).**

The Singleton pattern offers several benefits:

1. **Controlled access to sole instance**: The Singleton class encapsulates its sole instance, providing strict control over how and when clients access it.
2. **Reduced name space**: By using the Singleton pattern, you avoid polluting the namespace with global variables that store sole instances.
3. **Permits refinement of operations and representation**: The Singleton class can be subclassed, allowing for easy configuration of the application with an instance of the extended class at runtime.
4. **Permits a variable number of instances**: The pattern makes it easy to change the design to allow more than one instance of the Singleton class if needed. The same approach can be used to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.
5. **More flexible than class operations**: While another way to package a singleton's functionality is to use class operations (such as static member functions or class methods), these techniques make it difficult to change the design to allow more than one instance of a class. Moreover, static member functions in a class are compile-time, so subclasses can't override them polymorphically. The Singleton pattern provides more flexibility in this regard.

Implementation of Singleton

The implementation of the Singleton pattern in Java typically follows these steps:

1. **Add a private static field to the class for storing the singleton instance**:
    - This field holds the single instance of the class. It's declared as private static to ensure that only the class itself can access it.

```
public class Singleton {
    private static Singleton instance;
    // Other class members...
}
```

2. **Declare a public static creation method for getting the singleton instance**:
    - This method provides global access to the singleton instance. It's declared as public static so that it can be accessed without instantiating the class.

```
public class Singleton {
    private static Singleton instance;

    public static Singleton getInstance() {
        // Implementation of lazy initialization
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    // Other class members...
}
```

3. **Implement "lazy initialization" inside the static method**:
    - This ensures that the instance is created only when needed, i.e., on the first call to the `getInstance()` method. Subsequent calls return the same instance.

```
public class Singleton {
    private static Singleton instance;

    public static Singleton getInstance() {
        // Lazy initialization
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
```

```
    }
    // Other class members...
}
```

1. **Make the constructor of the class private**:
    - This prevents other classes from instantiating the Singleton class using the `new` keyword.

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        // Private constructor
    }

    public static Singleton getInstance() {
        // Lazy initialization
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
    // Other class members...
}
```

1. **Replace direct calls to the singleton's constructor with calls to its static creation method**:
    - Update any client code that directly calls the constructor of the Singleton class to call its `getInstance()` method instead.

```
// Client code
Singleton singleton = Singleton.getInstance();
```

Four methods of Singleton Instatiation

Method 1: Lazy Instantiation (Not Thread Safe)

```
class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- **Description**: Lazy instantiation ensures that the singleton object is created only when needed.
- **Problem**: Not thread-safe. If multiple threads try to access the `getInstance()` method simultaneously, it may result in the creation of multiple instances.

Method 2: Thread Synchronized

```
class Singleton {
    private static Singleton obj;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (obj == null) {
            obj = new Singleton();
        }
        return obj;
    }
}
```

- **Description**: Making the `getInstance()` method synchronized ensures that only one thread can execute it at a time, solving the thread safety issue.
- **Problem**: Synchronization can decrease performance, especially if `getInstance()` is called frequently.

Method 3: Eager Instantiation

```
class Singleton {
    private static Singleton obj = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return obj;
    }
}
```

- **Description**: The singleton instance is created when the class is loaded, ensuring thread safety.
- **Problem**: If the Singleton object is resource-intensive or not always needed, this method may not be efficient.

Method 4: Double Checked Locking (Best)

```
class Singleton {
    private static volatile Singleton instance;
    private String data;

    private Singleton(String data) {
    this.data = data;
    }

    public static Singleton getInstance(String data) {
            Singleton result = instance;
        if (result == null) {
            synchronized (Singleton.class) {
                    result = instance;
                if (result == null) {
                    instance = result = new Singleton();
                }
            }
        }
        return result;
```

```
        }
    }
```

- **Description**: This method uses double-checked locking to ensure thread safety and improve performance.
- **Advantage**: Reduces the overhead of synchronization by only synchronizing when necessary.
- **Note: The `volatile` keyword ensures that multiple threads handle the `obj` variable correctly when it is being initialized.**
- Volatile is used when there is parital construction of a constructor and this can cause issue **volatile reads directly from the main memory**
- `volatile`: The `volatile` keyword ensures that changes made by one thread to the `obj` variable are visible to other threads immediately. Without `volatile`, the Java Memory Model could allow the `obj` reference to be cached thread-locally, which would mean that one thread could modify `obj`, but other threads might not see the change, leading to inconsistent behavior.
- Use local varibales to improve memory by 40%

Each method has its own use case depending on the requirements of your application. Double-checked locking is generally considered the best approach as it provides both thread safety and good performance.

Pros	Cons
1. You can be sure that a class has only a single instance.	1. Violates the Single Responsibility Principle. The pattern solves two problems at the time.
2. You gain a global access point to that instance.	2. The Singleton pattern can mask bad design, for instance, when the components of the program know too much about each other.
3. The singleton object is initialized only when it's requested for the first time.	3. The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
	4. It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects.

Factory Design Pattern (Virtual Constructor)

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

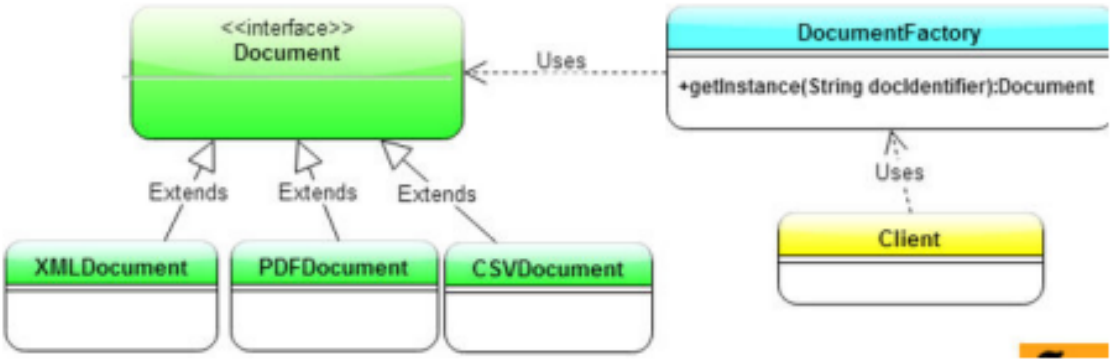
- Loosens the coupling of a given code by separation of products's construction code form the code that uses the product
- This pattern delegates the responsibility of initializing a class from the client to a particular factory class by creating a type of virtual constructor.
- Relies heavily on inheritence

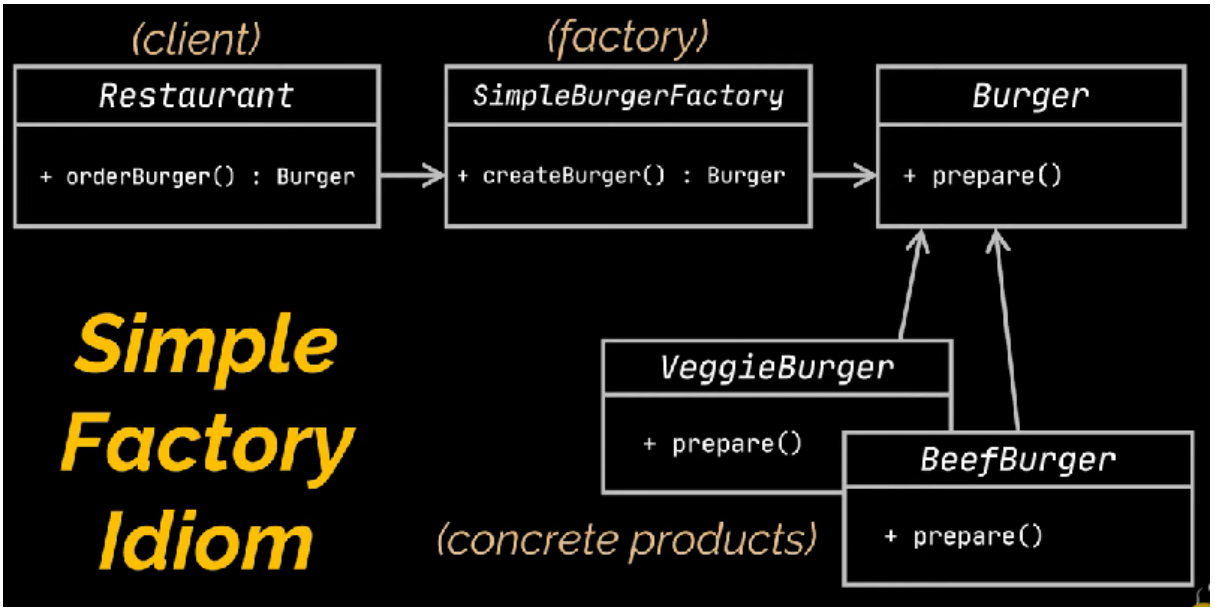
Intent

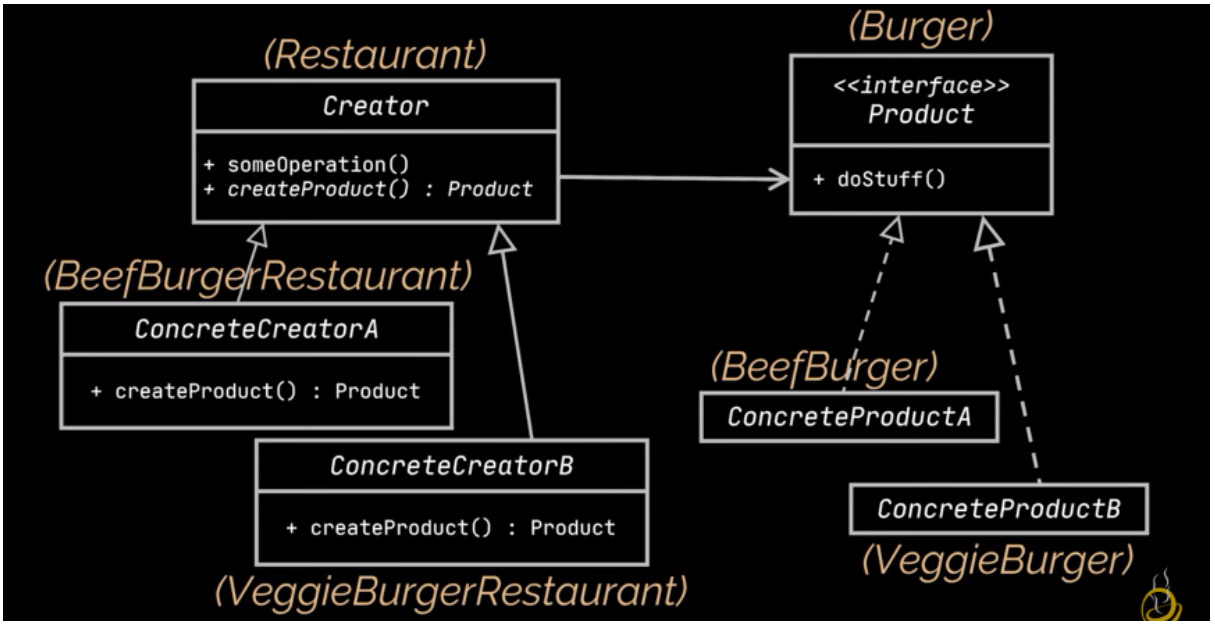
To deal with the problem of creating objects without having to specify the exact class of the object that will be created.

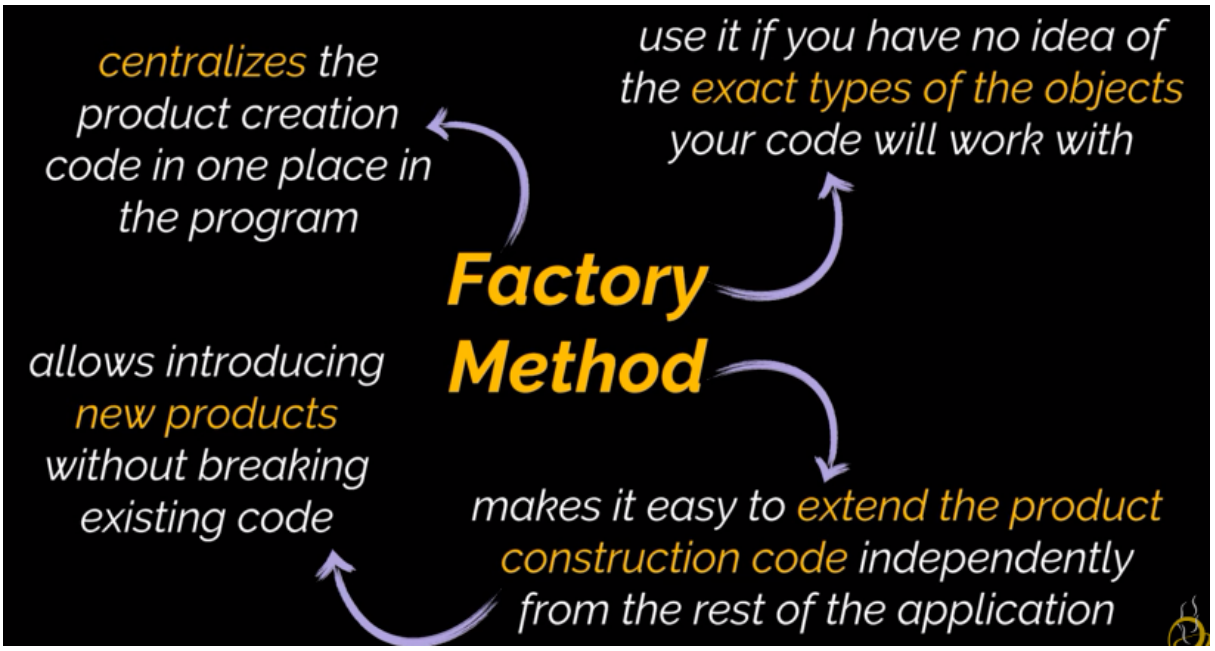
Motivation

- Creating an object often requires complex processes not appropriate to include within a composing object.
- The object's creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object's concerns.
- The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.


<<interface>>
Document
Uses
DocumentFactory
+getInstance(String docIdentifier):Document
Extends
Extends
Extends
Uses
Client
XMLDocument
PDFDocument
CSVDocument


(client)
(factory)
Restaurant
+ orderBurger() : Burger
SimpleBurgerFactory
+ createBurger() : Burger
Burger
+ prepare()
Simple Factory Idiom
VeggieBurger
+ prepare()
BeefBurger
+ prepare()
(concrete products)





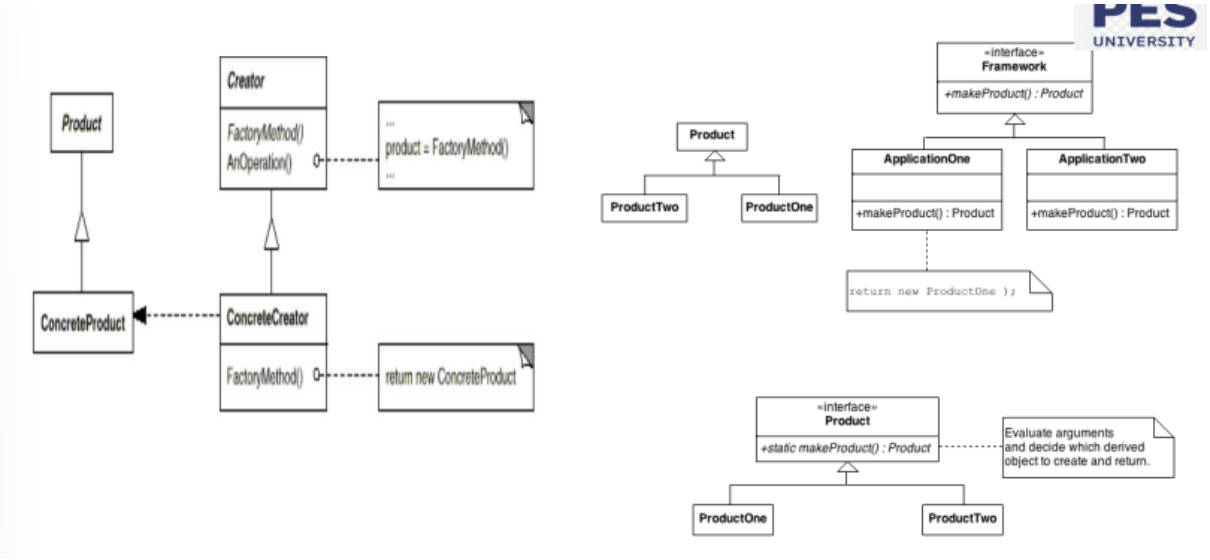
The Factory pattern is used when:

- **A class can't anticipate the class of objects it must create.**
  - In situations where a class needs to create objects, but the exact class of the objects may vary at runtime, the Factory pattern can be used. The class delegates the responsibility of object creation to a factory method or a factory class.
- **A class wants its subclasses to specify the objects it creates.**
  - When a class wants its subclasses to specify the objects it creates, it can define a factory method that the subclasses can override to create

the appropriate objects.

- **Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.**
    - When a class delegates the responsibility of object creation to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate, the Factory pattern can be used to provide a centralized way of creating objects.

The Factory pattern helps in decoupling the client code from the concrete classes it uses, thus making the code more flexible and easier to maintain. It also provides a way to encapsulate object creation logic, making the code more modular and easier to extend.



Participants:

- **Product**:
    - Defines the interface of objects the factory method creates.
- **ConcreteProduct**:
    - Implements the `Product` interface.
- **Creator**:
    - Declares the factory method, which returns an object of type `Product`.
    - May also define a default implementation of the factory method that returns a default `ConcreteProduct` object.

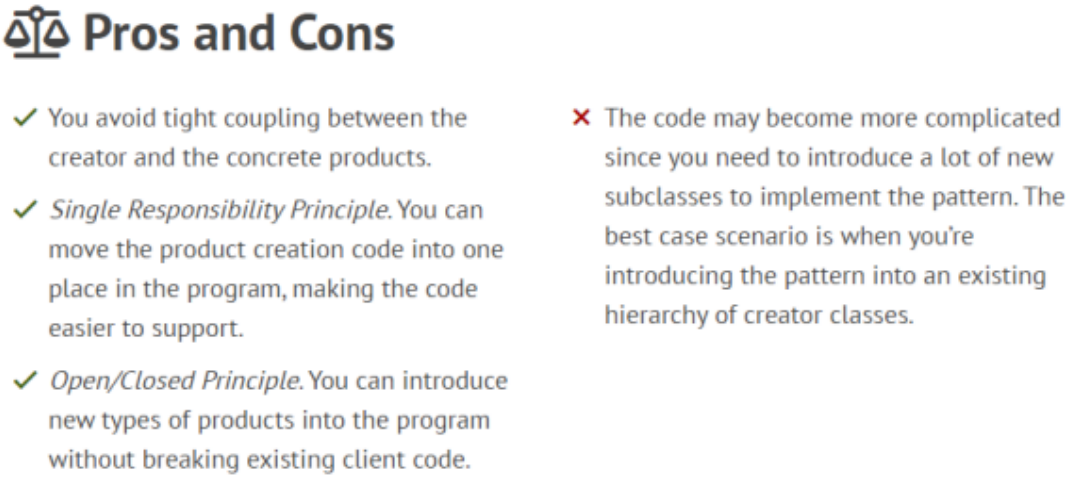
    - May call the factory method to create a `Product` object.
- **ConcreteCreator**:
    - Overrides the factory method to return an instance of a `ConcreteProduct`.

These participants collaborate to implement the Factory pattern. Here's how they work together:

- **The `Creator` declares the factory method, which returns a `Product` object.**
- **The `ConcreteCreator` overrides the factory method to return an instance of a specific `ConcreteProduct`.**
- **The `Product` interface defines the methods that all `ConcreteProduct` objects must implement.**
- **The `ConcreteProduct` implements the `Product` interface, providing the concrete implementation of the product.**

Consequences:

1. **Provides hooks for subclasses**:
    - Creating objects inside a class with a factory method is always more flexible than creating an object directly.
    - Factory Method gives subclasses a hook for providing an extended version of an object.
2. **Connects parallel class hierarchies**:
    - Factory methods are not only called by Creators, but can also be used by clients, especially in the case of parallel class hierarchies.
    - Parallel class hierarchies arise when a class delegates some of its responsibilities to a separate class.
    - For example, in graphical applications, graphical figures can be manipulated interactively. Implementing such interactions often requires storing and updating information that records the state of the manipulation at a given time. Factory methods can help manage these parallel class hierarchies efficiently.

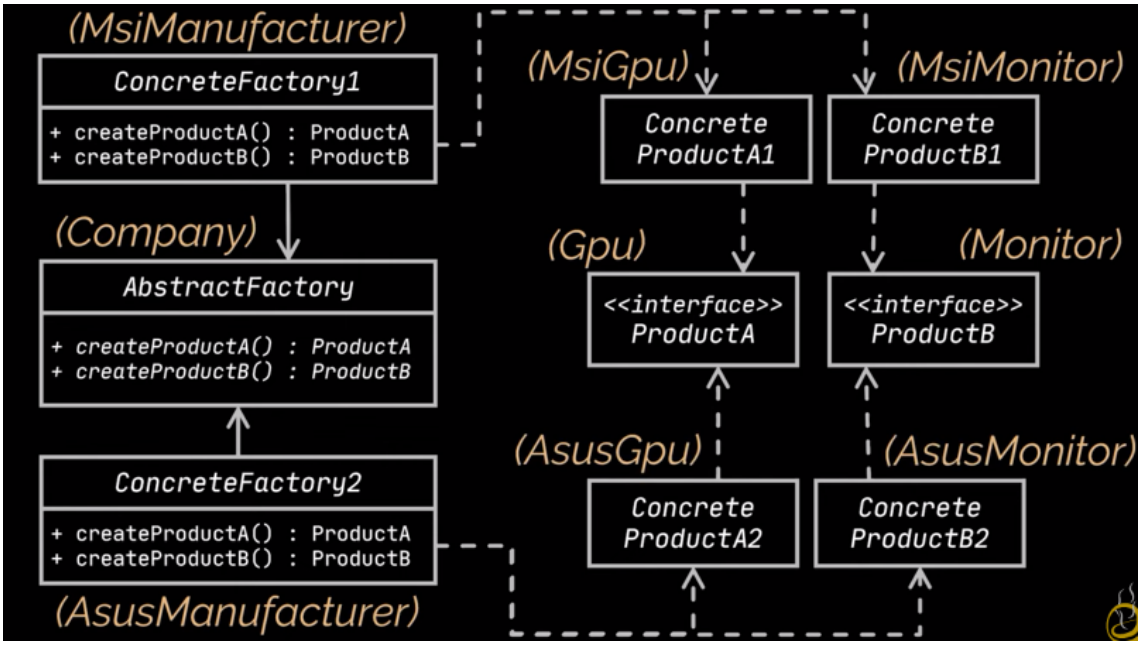


Issues to consider when using the Factory pattern:

1. **Two major varieties**:
   - There are two main variations of the Factory Method pattern:
     - When the `Creator` class is an abstract class and does not provide an implementation for the factory method it declares.
     - When the `Creator` is a concrete class and provides a default implementation for the factory method.
   - It's also possible to have an abstract class that defines a default implementation, but this is less common.
   - In the first case, subclasses must define an implementation because there's no reasonable default.
2. **Parameterized factory methods**:
   - Another variation on the pattern lets the factory method create multiple kinds of products.
   - The factory method takes a parameter that identifies the kind of object to create.
   - All objects the factory method creates will share the `Product` interface.
3. **Language-specific variants and issues**:
   - Different languages have their own interesting variations and caveats.

- In Smalltalk, for example, programs often use a method that returns the class of the object to be instantiated.
- A `Creator` factory method can use this value to create a product, and a `ConcreteCreator` may store or even compute this value.
- The result is an even later binding for the type of `ConcreteProduct` to be instantiated.

Abstract Factory

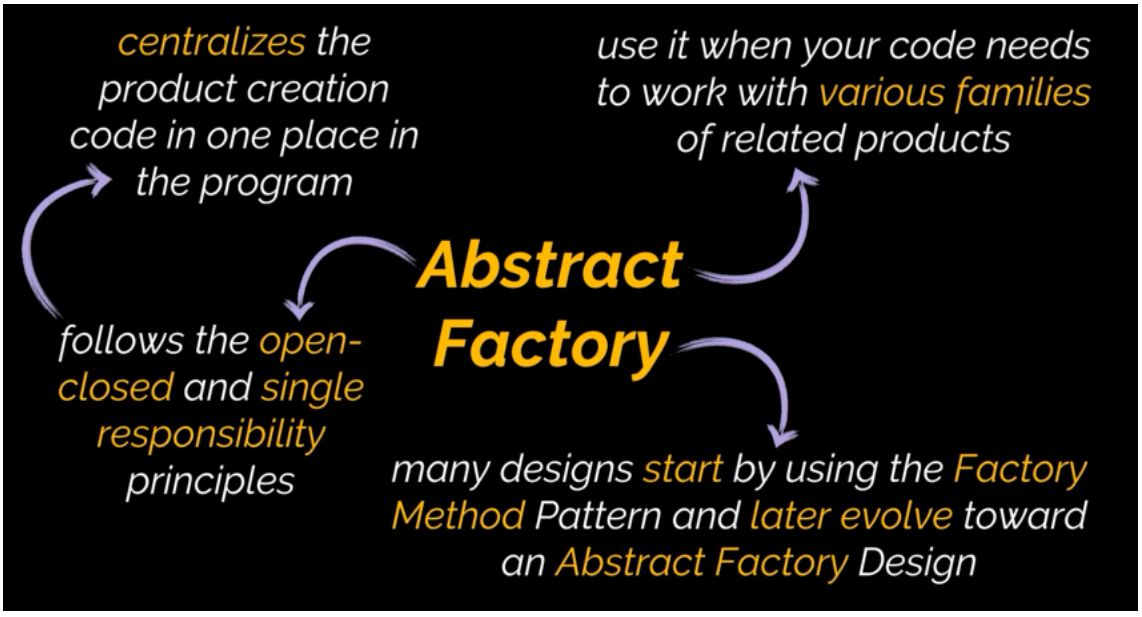


Creator (Abstract Factory) : Instantiaties objects of products based on concreateFactory (Car Manufacturer)

ConcreateFactory : Creates concreteProducts (Particular Car)

ConcreteProducts : Creates individual parts called Products (Parts of car)

Product : Most basic method Implementation (Implementation of those car parts)



Factory always try to follow Single Responsibility Principle and Open Closed Principle

Example

Participants:

- **Abstract Factory (FurnitureFactory)**:
  - Interface for creating abstract products (Chair and Table).
- **Concrete Factory (ModernFurnitureFactory, VictorianFurnitureFactory)**:
  - Implements the Abstract Factory interface.
  - Creates concrete products (ModernChair, ModernTable) or (VictorianChair, VictorianTable).
- **Abstract Products (Chair, Table)**:
  - Interface for the products.
- **Concrete Products (ModernChair, ModernTable, VictorianChair, VictorianTable)**:
  - Implements the Abstract Product interfaces.

Example:

```
// Abstract Product: Chair
interface Chair {
    void sitOn();
}

// Concrete Product: ModernChair
class ModernChair implements Chair {
    public void sitOn() {
        System.out.println("Sitting on a modern chair");
    }
}

// Concrete Product: VictorianChair
class VictorianChair implements Chair {
    public void sitOn() {
        System.out.println("Sitting on a victorian chair");
    }
}

// Abstract Product: Table
interface Table {
    void putOn();
}

// Concrete Product: ModernTable
class ModernTable implements Table {
    public void putOn() {
        System.out.println("Putting something on a modern t
able");
    }
}

// Concrete Product: VictorianTable
class VictorianTable implements Table {
    public void putOn() {
        System.out.println("Putting something on a victoria
n table");
    }
```

```
}

// Abstract Factory
interface FurnitureFactory {
    Chair createChair();
    Table createTable();
}

// Concrete Factory: ModernFurnitureFactory
class ModernFurnitureFactory implements FurnitureFactory {
    public Chair createChair() {
        return new ModernChair();
    }

    public Table createTable() {
        return new ModernTable();
    }
}

// Concrete Factory: VictorianFurnitureFactory
class VictorianFurnitureFactory implements FurnitureFactory
{
    public Chair createChair() {
        return new VictorianChair();
    }

    public Table createTable() {
        return new VictorianTable();
    }
}

public class AbstractFactoryPatternExample {
    public static void main(String[] args) {
        // Create a modern furniture factory
        FurnitureFactory modernFactory = new ModernFurnitur
eFactory();

        // Create modern chair
```

```
        Chair modernChair = modernFactory.createChair();
        modernChair.sitOn();

        // Create modern table
        Table modernTable = modernFactory.createTable();
        modernTable.putOn();

        // Create a victorian furniture factory
        FurnitureFactory victorianFactory = new VictorianFu
rnitureFactory();

        // Create victorian chair
        Chair victorianChair = victorianFactory.createChair
();
        victorianChair.sitOn();

        // Create victorian table
        Table victorianTable = victorianFactory.createTable
();
        victorianTable.putOn();
    }
}
```

In this example:

- **Abstract Factory**: `FurnitureFactory`
- **Concrete Factory**: `ModernFurnitureFactory`, `VictorianFurnitureFactory`
- **Abstract Products**: `Chair`, `Table`
- **Concrete Products**: `ModernChair`, `ModernTable`, `VictorianChair`, `VictorianTable`

Builder Pattern

- Builder Design Pattern as it was intended implies that a sequence of complex operations is needed in order to produce an object instance.
- The idea is that we delegate the construction of an object to a specialized class, which is aware of the complex process and logic required to build that specific instance.

- This helps us create Single Responsibility classes for complex object creation while at the same time ensuring separation of object creation from business logic.
- Based on the nature of the application, Builder implementations might lead to code re-usability, reducing the code base and improving SOLID compliance of our code.

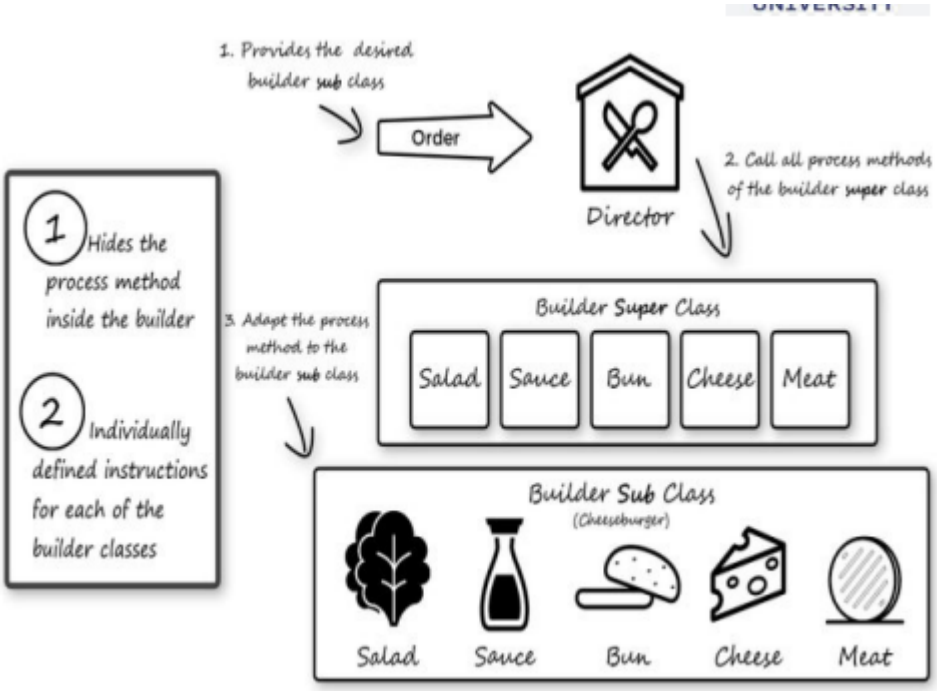
Builder Pattern Components:

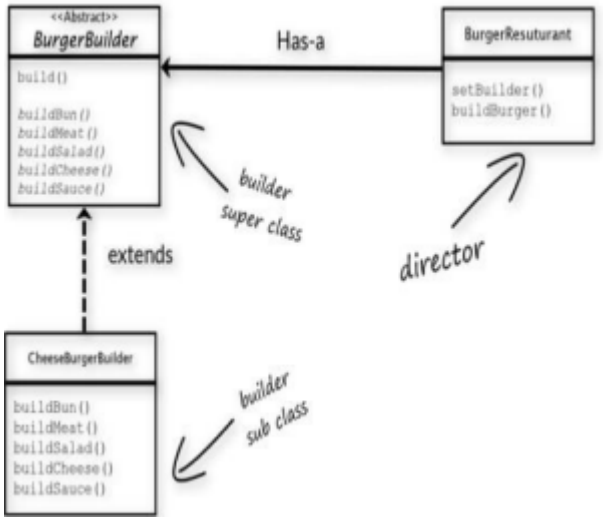
When talking about the Builder design pattern, it is important to understand the concept of the **Director** and the **Builder**.

- **Director**:
  - The Director's job is to invoke the building process of the builder.
  - It coordinates the construction process by directing the builder.
- **Builder**:
  - The Builder's job is to manage the different building procedures associated with each of the different variations of objects.
  - In the context of burgers, for example, the Builder manages the construction of different types of burgers.

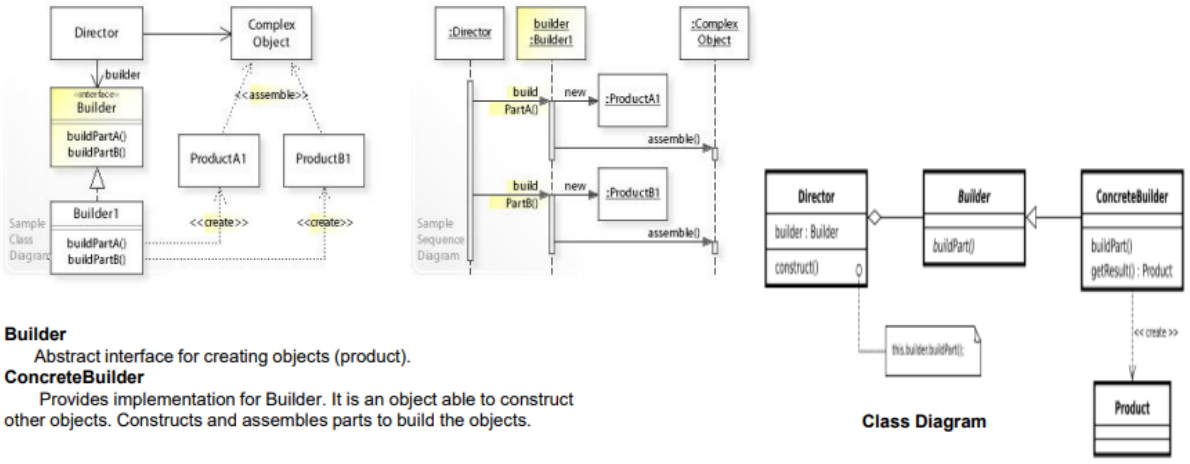
Builder Pattern Structure:

- The Builder pattern consists of two main class types:
  - **Builder**:
    - Defines an abstract interface for creating parts of a complex object.
    - Provides methods for building the various parts of the complex object.
  - **Director**:
    - Constructs an object using the Builder interface.
    - Directs the building process.





UML class diagram for the Builder Pattern



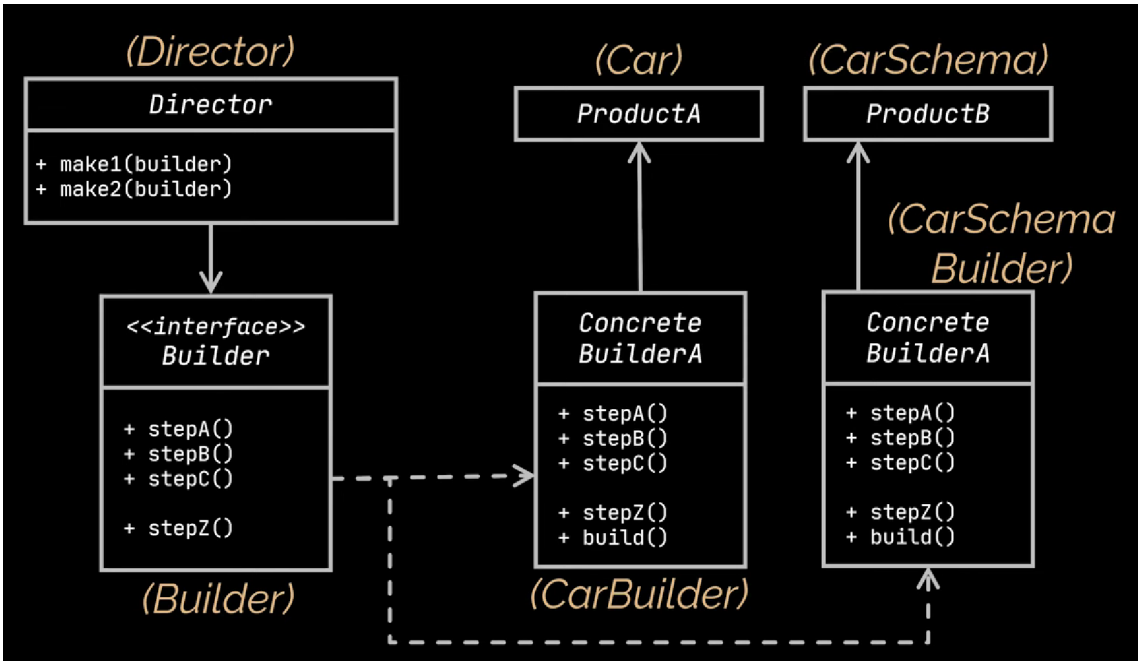
Builder
Abstract interface for creating objects (product).
ConcreteBuilder
Provides implementation for Builder. It is an object able to construct other objects. Constructs and assembles parts to build the objects.

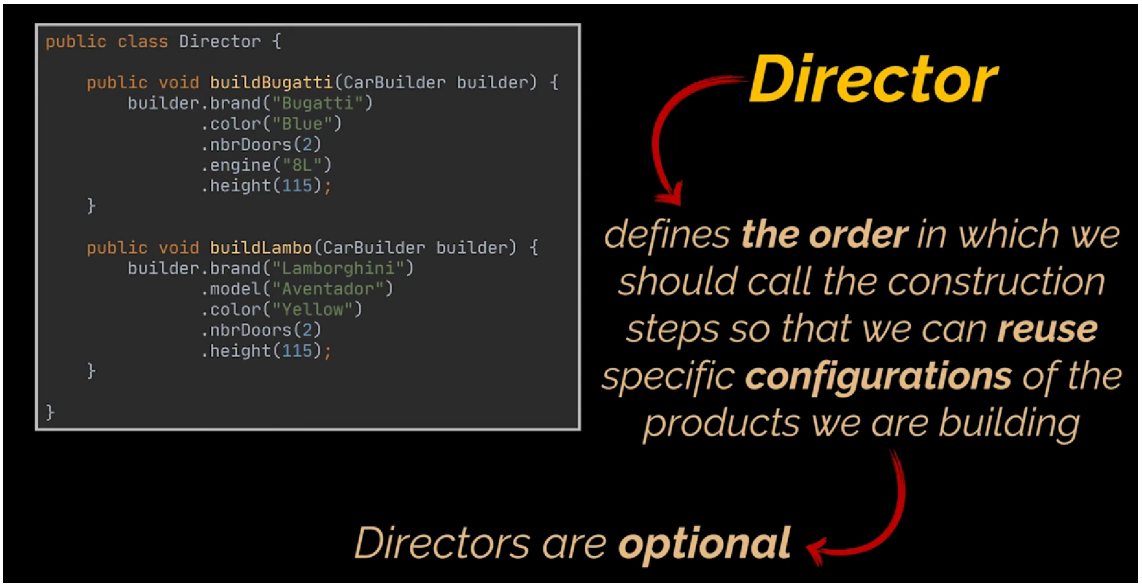
Class Diagram

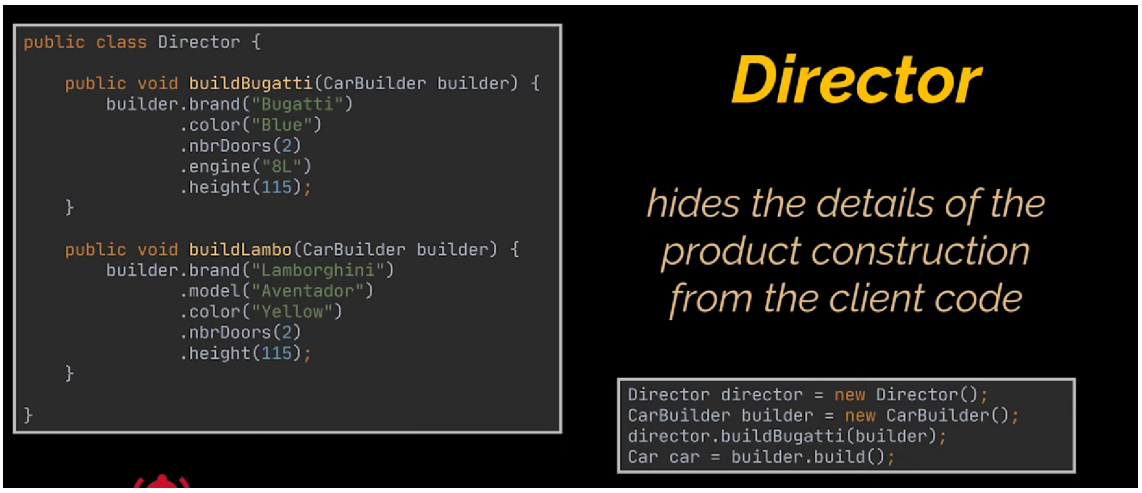
```java
public class CarBuilder {
    private int id;
    private String brand;
    private String model;
    private String color;
    ...

    public CarBuilder id(int id) {
        this.id = id;
    }

    public CarBuilder brand(String brand) {
        this.brand = brand;
    }

    public CarBuilder model(String model) {
        this.model = model;
    }

    public CarBuilder color(String color) {
        this.color = color;
    }
    ...

    public Car build() {
        return new Car(id, brand, model, color);
    }
}
```

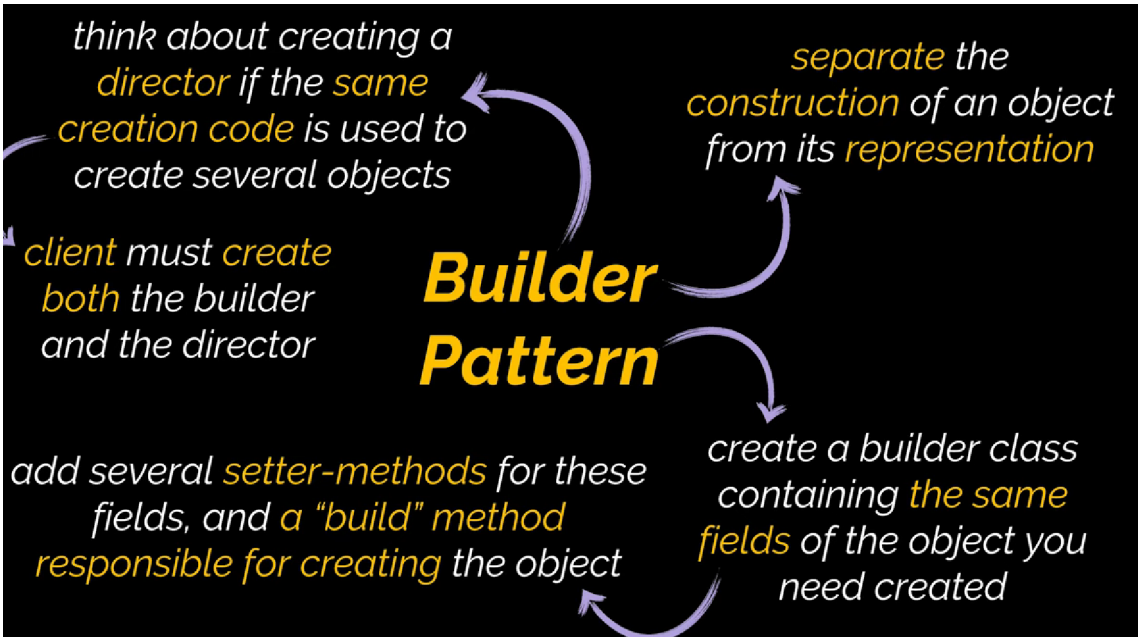
```java
public class Car {
    private final int id;
    private final String brand;
    private final String model;
    private final String color;
    ...

    Car(int id, String brand, String model,
                                String color, ...) {
        this.id = id;
        this.brand = brand;
        this.model = model;
        this.color = color;
        ...
    }
}
```

```java
CarBuilder builder = new CarBuilder();
builder.id(2122)
       .brand("Bugatti")
       .model("Chiron")
       .color("Blue");
Car car = Builder.build();
```



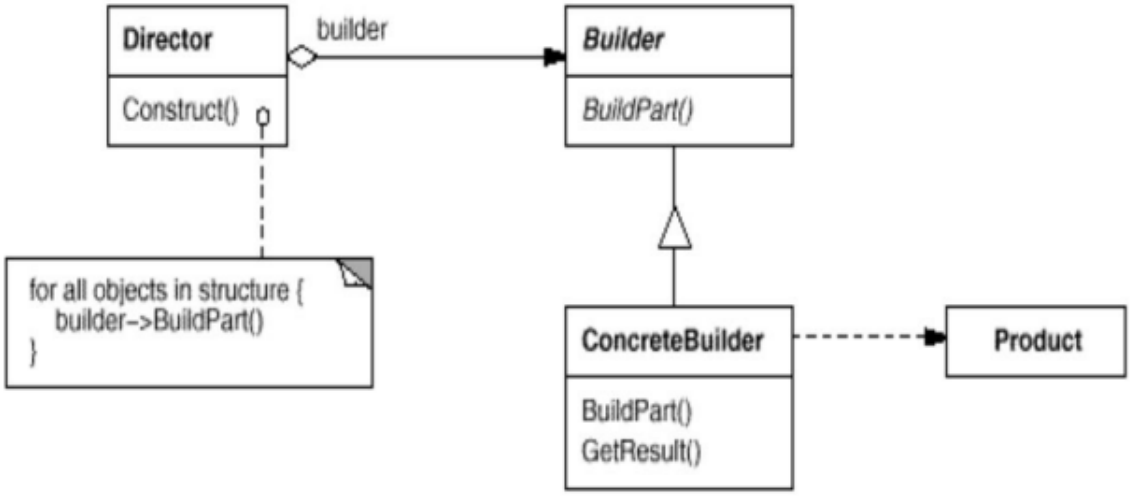

public class Director {
public void buildBugatti(CarBuilder builder) {
builder.brand("Bugatti")
.color("Blue")
.nbrDoors(2)
.engine("8L")
.height(115);
}
public void buildLambo(CarBuilder builder) {
builder.brand("Lamborghini")
.model("Aventador")
.color("Yellow")
.nbrDoors(2)
.height(115);
}
}
Director
defines the order in which we should call the construction steps so that we can reuse specific configurations of the products we are building
Directors are optional


public class Director {
public void buildBugatti(CarBuilder builder) {
builder.brand("Bugatti")
.color("Blue")
.nbrDoors(2)
.engine("8L")
.height(115);
}
public void buildLambo(CarBuilder builder) {
builder.brand("Lamborghini")
.model("Aventador")
.color("Yellow")
.nbrDoors(2)
.height(115);
}
}
Director
hides the details of the product construction from the client code
Director director = new Director();
CarBuilder builder = new CarBuilder();
director.buildBugatti(builder);
Car car = builder.build();


think about creating a director if the same creation code is used to create several objects
separate the construction of an object from its representation
client must create both the builder and the director
Builder Pattern
add several setter-methods for these fields, and a "build" method responsible for creating the object
create a builder class containing the same fields of the object you need created

Applicability of the Builder Pattern:

- **The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled**:
  - The Builder pattern separates the construction of a complex object from its representation. It allows the same construction process to create different representations.
- **The construction process must allow different representations for the object that's constructed**:
  - The Builder pattern allows you to vary a product's internal representation. It hides the product's construction details from the client.
- **You want to get rid of a "telescoping constructor"**:
  - When you have a constructor with many optional parameters, it's inconvenient to call. The Builder pattern solves this problem by providing a cleaner way to construct objects with many parameters.
- **You want your code to be able to create different representations of some product**:
  - For example, stone and wooden houses. The Builder pattern can be applied when the construction of various representations of the product involves similar steps that differ only in the details.



Participants in the Builder Pattern:

- **Builder**:
  - Specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder** :
  - Constructs and assembles parts of the product by implementing the Builder interface.
  - Defines and keeps track of the representation it creates.
  - Provides an interface for retrieving the product
- **Director**
  - Constructs an object using the Builder interface.
- **Product**:
  - Represents the complex object under construction.
  - ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
  - Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Collaboration

- The **Director** is responsible for coordinating the construction process.
- The **Director** decides what product it wants to build and provides instructions to the **Builder** interface.
- The **Builder** interface declares methods for building parts of the product.
- The **ConcreteBuilder** implements the **Builder** interface and provides implementations for the methods declared in the interface.

Example

Participants:

- **Builder (HouseBuilder)**:
  - Interface for creating parts of a `House` object.
- **ConcreteBuilder (ConcreteHouseBuilder)**:
  - Implements the Builder interface.

- Constructs and assembles parts of the `House`.
- Provides a method for retrieving the constructed `House`.

- **Director (CivilEngineer)**:
  - Constructs a `House` object using the Builder interface.
- **Product (House)**:
  - Represents the complex object under construction.

Example:

```
// Product
class House {
    private String foundation;
    private String structure;
    private String roof;
    private String interior;

    public void setFoundation(String foundation) {
        this.foundation = foundation;
    }

    public void setStructure(String structure) {
        this.structure = structure;
    }

    public void setRoof(String roof) {
        this.roof = roof;
    }

    public void setInterior(String interior) {
        this.interior = interior;
    }

    @Override
    public String toString() {
        return "House{" +
                "foundation='" + foundation + '\\'' +
                ", structure='" + structure + '\\'' +
```

```
                ", roof='" + roof + '\\'' +
                ", interior='" + interior + '\\'' +
                '}';
    }
}

// Builder
interface HouseBuilder {
    void buildFoundation();
    void buildStructure();
    void buildRoof();
    void buildInterior();
    House getResult();
}

// ConcreteBuilder
class ConcreteHouseBuilder implements HouseBuilder {
    private House house;

    public ConcreteHouseBuilder() {
        this.house = new House();
    }

    public void buildFoundation() {
        house.setFoundation("Concrete Foundation");
    }

    public void buildStructure() {
        house.setStructure("Concrete Walls");
    }

    public void buildRoof() {
        house.setRoof("Concrete Roof");
    }

    public void buildInterior() {
        house.setInterior("Concrete Interior");
    }
```

```
    public House getResult() {
        return house;
    }
}

// Director
class CivilEngineer {
    private HouseBuilder houseBuilder;

    public CivilEngineer(HouseBuilder houseBuilder) {
        this.houseBuilder = houseBuilder;
    }

    public void constructHouse() {
        houseBuilder.buildFoundation();
        houseBuilder.buildStructure();
        houseBuilder.buildRoof();
        houseBuilder.buildInterior();
    }

    public House getHouse() {
        return houseBuilder.getResult();
    }
}

public class BuilderPatternExample {
    public static void main(String[] args) {
        // Create ConcreteBuilder
        HouseBuilder houseBuilder = new ConcreteHouseBuilde
r();

        // Create Director
        CivilEngineer civilEngineer = new CivilEngineer(hou
seBuilder);

        // Construct House
        civilEngineer.constructHouse();
```

```
        // Get the result from ConcreteBuilder
        House house = civilEngineer.getHouse();

        // Print the result
        System.out.println(house);
    }
}
```

In this example:

- **Builder**: `HouseBuilder`
- **ConcreteBuilder**: `ConcreteHouseBuilder`
- **Director**: `CivilEngineer`
- **Product**: `House`

Consequences of the Builder Pattern:

1. **Lets you vary a product's internal representation by using different Builders**:
    - With the Builder pattern, you can create different representations of a product by using different builders.
    - This allows you to vary the internal structure of the product without changing its representation.
2. **Isolates code for construction and representation**:
    - The Builder pattern separates the construction of a complex object from its representation.
    - This isolation ensures that the construction code is independent of the specific types of objects that are being constructed.
3. **Gives finer-grain control over the construction process**:
    - The Builder pattern allows you to have finer-grain control over the construction process.
    - It provides a step-by-step approach to construction, allowing you to specify the construction process in detail.

Issues to consider when using the Builder pattern:

1. **Assembly and construction interface: generality**:
    - The interface for constructing objects should be general enough to accommodate various implementations.
    - The Builder pattern should provide a flexible way to construct objects without making the interface too complex or specific.
2. **Is an abstract class for all Products necessary?**:
    - Consider whether an abstract class for all products is necessary or if individual product classes suffice.
    - Determine if there's a common interface that all products must implement.
3. **Usually products don’t have a common interface**:
    - In many cases, products don't share a common interface.
    - The Builder pattern should allow for the construction of different types of products without imposing a common interface.
4. **Usually there’s an abstract Builder class that defines an operation for each component that a director may ask it to create**:
    - The Builder pattern often involves an abstract Builder class that defines operations for creating different components of a product.
    - ConcreteBuilder classes then implement these operations to construct specific products.
5. **These operations do nothing by default (empty, static methods)**:
    - The abstract Builder class may define default or empty methods for constructing product components.
    - ConcreteBuilder classes override these methods to provide specific implementations.
6. **The ConcreteBuilder overrides operations selectively**:
    - ConcreteBuilder classes selectively override operations defined in the abstract Builder class to construct specific products.

How to Implement the Builder Pattern:

1. **Define common construction steps**:
    - Clearly define the common construction steps for building all available product representations.
    - Declare these steps in the base builder interface.
2. **Create concrete builder classes**:
    - Create a concrete builder class for each of the product representations.
    - Implement the construction steps defined in the builder interface for each product representation.
3. **Consider creating a director class**:
    - The director class may encapsulate various ways to construct a product using the same builder object.
4. **Client code implementation**:
    - The client code creates both the builder and the director objects.
    - Before construction starts, the client must pass a builder object to the director.
5. **Retrieve the construction result**:
    - If all products follow the same interface, the construction result can be obtained directly from the director.
    - Otherwise, the client should fetch the result from the builder.

Pros

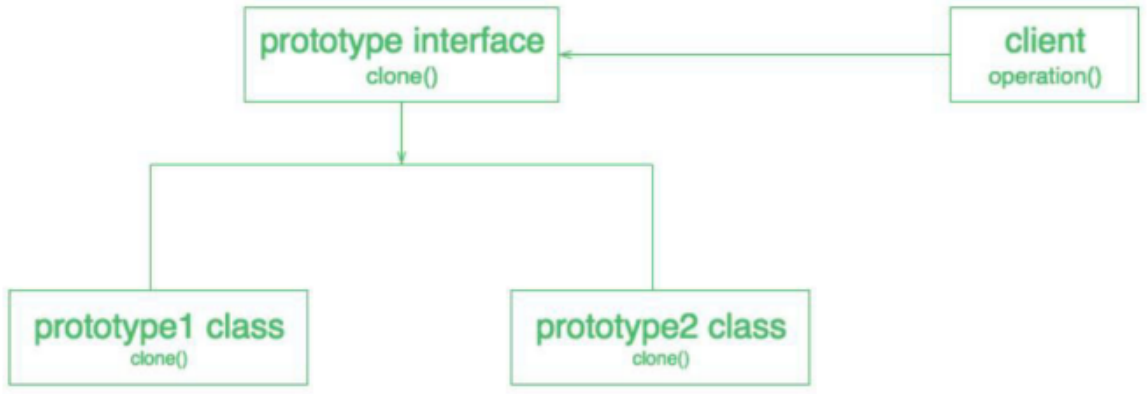
- ❑ You can construct objects step-by-step, defer construction steps or run steps recursively.
- ❑ You can reuse the same construction code when building various representations of products.
- ❑ *Single Responsibility Principle*. You can isolate complex construction code from the business logic of the product.

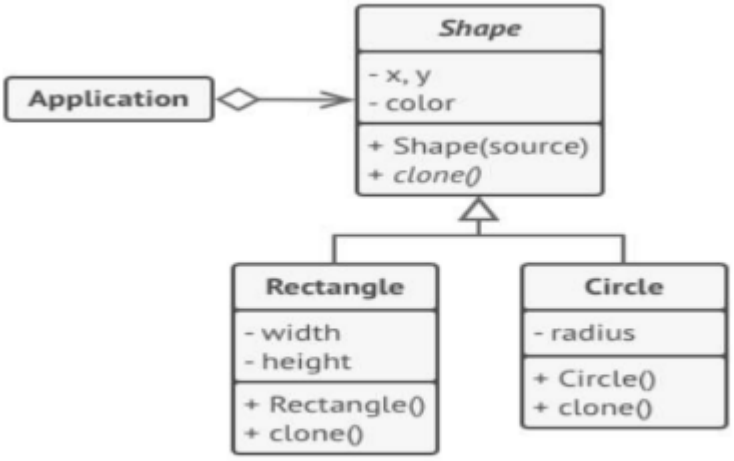
Cons

- ❑ The overall complexity of the code increases since the pattern requires creating multiple new classes.

Prototype Design Pattern

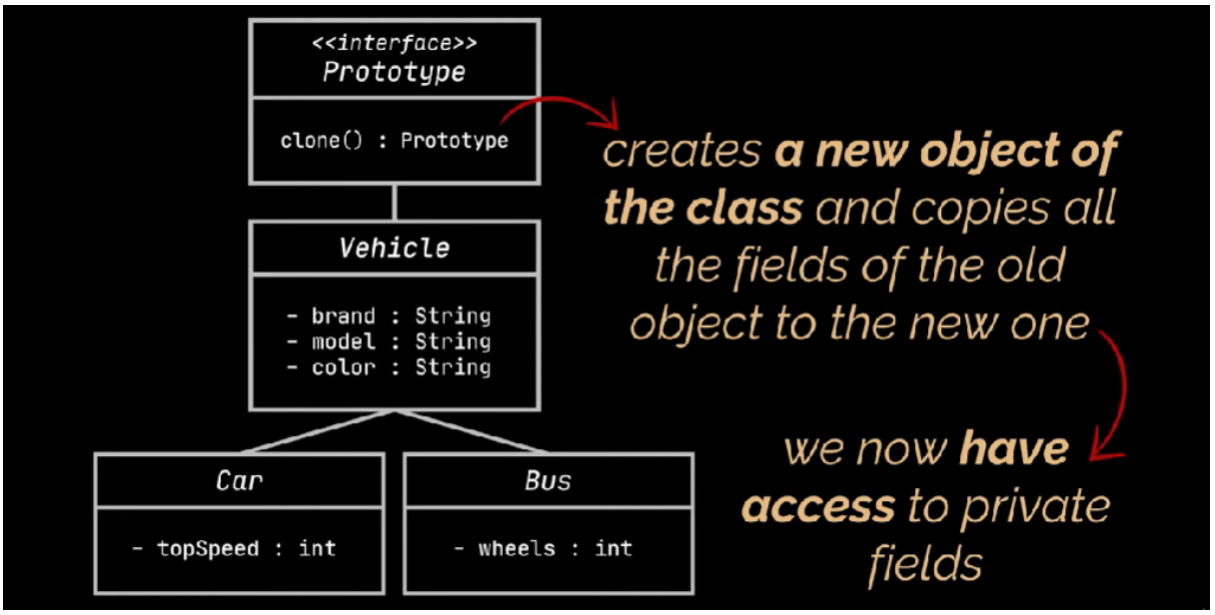
- Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.
- The Prototype Pattern specify the kind of objects to create using a prototypical instance, and create new objects by copying this prototype.
- The Prototype class code doesnt depend on the concrete class implementation
- Client code can make new instances without knowing which specific class is being instantiated
- For Java to use prototype pattern java.lang.Object#clone() (class should implement the java.lang.Cloneable interface)

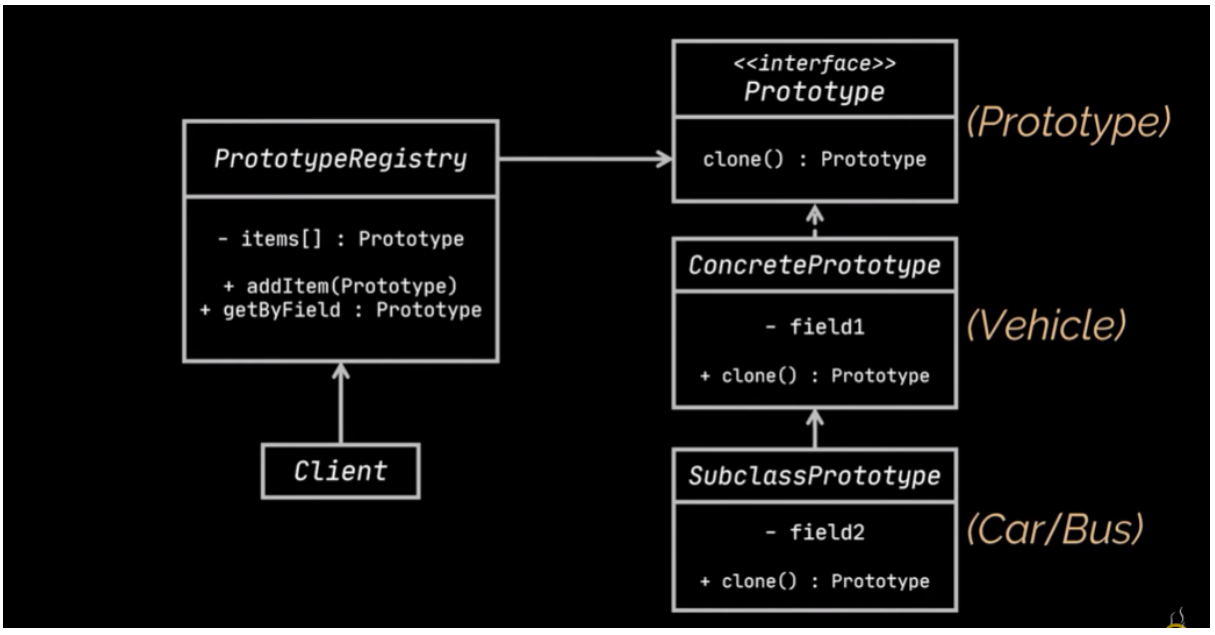




Cloning a set of objects that belong to a class hierarchy.

All shape classes follow the same interface, which provides a cloning method. A subclass may call the parent's cloning method before copying its own field values to the resulting object





this newly created car will reference the **same** 'GpsSystem' object in the memory, and any **change** done on this object **will be reflected in both** cars

Shallow Copy

```
public class Car extends Vehicle {
    private int topSpeed;
    private GpsSystem gpsSystem;

    public Car() {  }

    public Car(Car car) {
        super(car);
        this.topSpeed = car.topSpeed;
        this.gpsSystem = car.gpsSystem;
    }

    @Override
    public Car clone() {
        return new Car(this);
    }
}
```

```
public class GpsSystem {

}
```

any **change done** on the first 'GpsSystem' object **will not affect** the second one and vice versa

Deep Copy

```
public class Car extends Vehicle {
    private int topSpeed;
    private GpsSystem gpsSystem;

    public Car() {  }

    public Car(Car car) {
        super(car);
        this.topSpeed = car.topSpeed;
        this.gpsSystem = new GpsSystem();
    }           // or gpsSystem.clone();

    @Override
    public Car clone() {
        return new Car(this);
    }
}
```

```
public class GpsSystem {

}
```

use the Prototype Pattern when your code shouldn't depend on the concrete classes of the objects that you need to copy or duplicate

Prototype Pattern

will save resources and time

will clone objects without coupling them to their concrete classes

will get rid of repeated initialization code

Prototype Registry

In the Prototype Registry pattern, a centralized registry (or factory) is used to contain a set of pre-defined prototype objects. Instead of creating new objects directly, the factory retrieves them from the registry by passing their names or other parameters. The factory then searches for an appropriate prototype, clones it, and returns a copy.

Explanation:

1. **Centralized Registry (or Factory)**:
    - The factory maintains a centralized registry of pre-defined prototype objects.
2. **Pre-defined Prototype Objects**:
    - The registry contains a set of pre-defined prototype objects.
3. **Retrieving Objects from the Registry**:
    - To obtain a new object, the client requests it from the factory by passing its name or other parameters.
4. **Cloning Prototype Objects**:
    - The factory searches for the appropriate prototype object in the registry.
    - Once found, the factory clones the prototype object.
5. **Returning Cloned Copy**:
    - The factory returns a cloned copy of the prototype object to the client.

```
import java.util.HashMap;
import java.util.Map;

// Prototype interface
interface Prototype {
    Prototype clone();
}

// Concrete prototype: Circle
class Circle implements Prototype {
    private String color;
```

```
    private int radius;

    public Circle(String color, int radius) {
        this.color = color;
        this.radius = radius;
    }

    public Prototype clone() {
        return new Circle(color, radius);
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public int getRadius() {
        return radius;
    }

    public void setRadius(int radius) {
        this.radius = radius;
    }

    public String toString() {
        return "Circle [color=" + color + ", radius=" + rad
ius + "]";
    }
}

// Prototype registry (factory)
class PrototypeFactory {
    private static Map<String, Prototype> registry = new Ha
shMap<>();
```

```
    static {
        registry.put("redCircle", new Circle("red", 10));
        registry.put("blueCircle", new Circle("blue", 20));
    }

    public static Prototype createPrototype(String type) {
        return registry.get(type).clone();
    }
}

public class PrototypeRegistryExample {
    public static void main(String[] args) {
        Prototype redCircle = PrototypeFactory.createPrototot
ype("redCircle");
        Prototype blueCircle = PrototypeFactory.createProto
type("blueCircle");

        System.out.println(redCircle);
        System.out.println(blueCircle);
    }
}
```

In this example:

- We have a `Circle` class that implements the `Prototype` interface and represents a concrete prototype.
- We have a `PrototypeFactory` class that acts as a prototype registry or factory, which contains pre-defined prototype objects.
- We can retrieve new objects from the factory by passing their name (or other parameters) and get a copy of the prototype object.

Applicability of the Prototype Pattern:

Use the Prototype pattern when:

- **System should be independent of how its products are created, composed, and represented**:
    - The Prototype pattern allows a system to create new objects by cloning existing objects, making it independent of the concrete classes of

objects it needs to instantiate.

- **Classes to instantiate are specified at run-time**:
  - For example, by dynamic loading, when the exact class to instantiate is not known until run-time.
- **To avoid building a class hierarchy of factories that parallels the class hierarchy of products**:
  - Instead of creating a factory hierarchy to create different types of products, the Prototype pattern allows you to clone pre-existing prototypes.
- **Instances of a class can have only a few different combinations of state**:
  - It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Example:

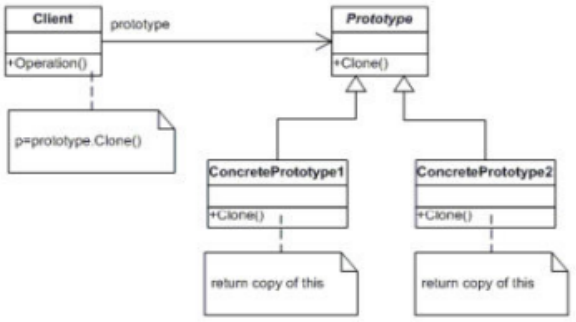
In a drawing application, the user can create different shapes (e.g., circles, squares, triangles). The exact type of shape to be created may not be known until run-time. Instead of building a factory hierarchy for creating different types of shapes, the application uses the Prototype pattern. It maintains a registry of pre-defined prototype objects (e.g., prototype circle, prototype square, prototype triangle). When the user requests a new shape, the application clones the appropriate prototype object and returns it. This way, the application is independent of the concrete classes of objects it needs to instantiate, and new shapes can be added to the system without modifying the application code.

Participants

Prototype: declares an interface for cloning itself.

ConcretePrototype: implements an operation for cloning itself.

Client: creates a new object by asking a prototype to clone itself and then making required modifications.



Collaboration:

- A client asks a prototype to clone itself.

Consequences:

Additional benefits of the Prototype pattern include:

- **Adding and removing products at run-time**:
  - New products can be added to the system by registering their prototypes in the prototype registry. Similarly, existing products can be removed from the system.
- **Specifying new objects by varying values**:
  - New objects can be created by varying the values of the existing prototypes. This allows for the creation of new objects with different attributes without creating new classes.
- **Specifying new objects by varying structure**:
  - New objects can be created by varying the structure of existing prototypes. This allows for the creation of new objects with different internal compositions without creating new classes.
- **Reduced subclassing**:
  - The Prototype pattern reduces the need for subclassing. Instead of creating subclasses for each variation of an object, the pattern allows for the creation of new objects by cloning existing prototypes.
- **Configuring an application with classes dynamically**:

- The Prototype pattern allows for the dynamic configuration of an application with classes at run-time. New classes can be registered as prototypes in the prototype registry, and objects of these classes can be created by cloning the prototypes as needed.

Issues to consider when using the Prototype pattern:

Consider the following issues when implementing prototypes:

- **Using a prototype manager**:
  - When the number of prototypes in a system isn't fixed, keep a registry of available prototypes. Clients won't manage prototypes themselves but will store and retrieve them from the registry. A client will ask the registry for a prototype before cloning it. We call this registry a prototype manager.
- **Implementing the Clone operation**:
  - The hardest part of the Prototype pattern is implementing the Clone operation correctly. It's particularly tricky when object structures contain circular references.
- **Initializing clones**:
  - While some clients are perfectly happy with the clone as is, others will want to initialize some or all of its internal state to values of their choosing.
  - You generally can't pass these values in the Clone operation because their number will vary between classes of prototypes. Some prototypes might need multiple initialization parameters; others won't need any.
  - Passing parameters in the Clone operation precludes a uniform cloning interface.

⚖ Pros and Cons

- ✓ You can clone objects without coupling to their concrete classes.
- ✓ You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- ✓ You can produce complex objects more conveniently.
- ✓ You get an alternative to inheritance when dealing with configuration presets for complex objects.
- ✗ Cloning complex objects that have circular references might be very tricky.

Example

```
import java.util.HashMap;
import java.util.Map;

// Prototype interface
interface Prototype {
    Prototype clone();
}

// Concrete prototype - Circle
class Circle implements Prototype {
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    public Prototype clone() {
        return new Circle(this.radius);
    }

    @Override
    public String toString() {
        return "Circle{radius=" + radius + '}';
```

```
    }
}

// Concrete prototype - Rectangle
class Rectangle implements Prototype {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public Prototype clone() {
        return new Rectangle(this.width, this.height);
    }

    @Override
    public String toString() {
        return "Rectangle{width=" + width + ", height=" + h
eight + '}';
    }
}

// Prototype registry (or manager)
class PrototypeManager {
    private static final Map<String, Prototype> prototypes
= new HashMap<>();

    static {
        prototypes.put("Circle", new Circle(10));
        prototypes.put("Rectangle", new Rectangle(5, 10));
    }

    public static Prototype getPrototype(String type) {
        Prototype prototype = prototypes.get(type);
        if (prototype != null) {
            return prototype.clone();
```

```
        }
        return null;
    }
}

public class PrototypePatternExample {
    public static void main(String[] args) {
        Prototype circle1 = PrototypeManager.getPrototype
("Circle");
        Prototype circle2 = PrototypeManager.getPrototype
("Circle");

        System.out.println(circle1);  // Output: Circle{rad
ius=10}
        System.out.println(circle2);  // Output: Circle{rad
ius=10}
        System.out.println(circle1 == circle2);  // Output:
false (different objects)

        Prototype rectangle = PrototypeManager.getPrototype
("Rectangle");
        System.out.println(rectangle);  // Output: Rectangl
e{width=5, height=10}
    }
}
```

In this example:

- `Prototype` is the interface implemented by all concrete prototypes.
- `Circle` and `Rectangle` are concrete prototype classes that implement the `Prototype` interface.
- `PrototypeManager` is the prototype registry (or manager) that stores pre-defined prototype objects (circle and rectangle).
- The client obtains prototype objects from the `PrototypeManager` by specifying their types ("Circle" or "Rectangle").
- Cloning of prototype objects is performed by calling the `clone()` method defined in the `Prototype` interface.

Client code can make new instances without knowing which specific class is being instantiated

Prototype pattern is used when the classes to instantiate are specified during run-time (Dynamic Loading)

Guideline on when to use each creational pattern:

Singleton Pattern:

- **Use Singleton when:**
  - You need to ensure that a class has only one instance, and that instance needs to be accessible globally.
  - You want to provide a global access point to that instance.
  - Examples:
    - Database connection classes.
    - Logger classes.
    - Configuration classes.

Factory Pattern:

- **Use Factory when:**
  - A class can't anticipate the class of objects it must create.
  - A class wants its subclasses to specify the objects it creates.
  - Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.
  - Examples:
    - GUI component factories.
    - Database driver factories.
    - LoggerFactory.

Builder Pattern:

- **Use Builder when:**
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
  - The construction process must allow different representations for the object that's constructed.
  - You want to get rid of a "telescoping constructor" or a constructor with too many parameters.
  - You want your code to be able to create different representations of some product.
  - Examples:
    - StringBuilder in Java.
    - DocumentBuilder in Java.
    - VehicleBuilder in a car manufacturing application.

Prototype Pattern:

- **Use Prototype when:**
  - A system should be independent of how its products are created, composed, and represented.
  - The classes to instantiate are specified at run-time, for example, by dynamic loading.
  - To avoid building a class hierarchy of factories that parallels the class hierarchy of products.
  - Instances of a class can have one of only a few different combinations of state.
  - Examples:
    - Cloning objects in a drawing application.
    - Cloning configuration objects.
    - Cloning documents in a word processor.

Summary:

- **Singleton:** For ensuring a class has only one instance and providing a global access point to that instance.
- **Factory:** For creating objects without specifying the exact class of object that will be created or when different subclasses may create different representations of an object.
- **Builder:** For creating complex objects with step-by-step approaches, especially when dealing with a large number of parameters.
- **Prototype:** For creating new objects by cloning existing objects, especially when the exact class of object to be created may not be known until run-time or when instances of a class can have only a few different combinations of state.