Syllabus:

Creational:
1. Factory
2. Abstract Factory
3. Singleton
4. Builder

Structural:
1. Adapter
2. Bridge
3. Decorator
4. Facade
5. Proxy

Behavioural:
1. Observer

Aspect	Creational Design Patterns	Structural Design Patterns	Behavioral Design Patterns
Intent	Focus on object creation mechanisms	Deal with the composition of classes and objects	Concentrate on the interaction between objects
Problem Addressed	Handling object creation complexities	Managing class and object composition	Managing object collaboration and responsibilities
Key Patterns	Singleton, Factory Method, Abstract Factory	Adapter, Decorator, Composite, Proxy	Observer, Strategy, Command, State
Flexibility	High level of flexibility in object creation	Enhances the flexibility of class composition	Improves flexibility in defining object interactions
Complexity	Addresses complexities of object creation	Addresses complexities of class and object structure	Addresses complexities in object collaboration
Example	Singleton ensures a class has only one instance	Adapter allows incompatible interfaces to work together	Observer defines a one-to-many dependency between objects
Common Use Cases	When a system should be configured with a single instance	When you want to use existing classes with incompatible interfaces	When you need to define communication between objects in a flexible way
UML Diagram	Often depicted with a single class and an arrow pointing to it	Typically represented with a structure diagram showing how classes and objects are composed	Often represented with a collaboration or sequence diagram showing how objects interact
Design Principle	Encapsulate object creation, promote code reuse	Favor composition over inheritance, adapt to varying interfaces	Identify the responsibilities and define the collaboration between objects
Examples	Singleton, Factory Method, Abstract Factory	Adapter, Decorator, Composite, Proxy	Observer, Strategy, Command, State

Factory Pattern :

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

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

Advantages of Factory Design Pattern

- Factory Method Pattern allows the sub-classes to choose the type of objects to create.
- Loose Coupling: Reduces dependencies between client code and concrete classes.
- Flexibility: Easily add new object types without breaking existing code.
- Promotes code reusability.

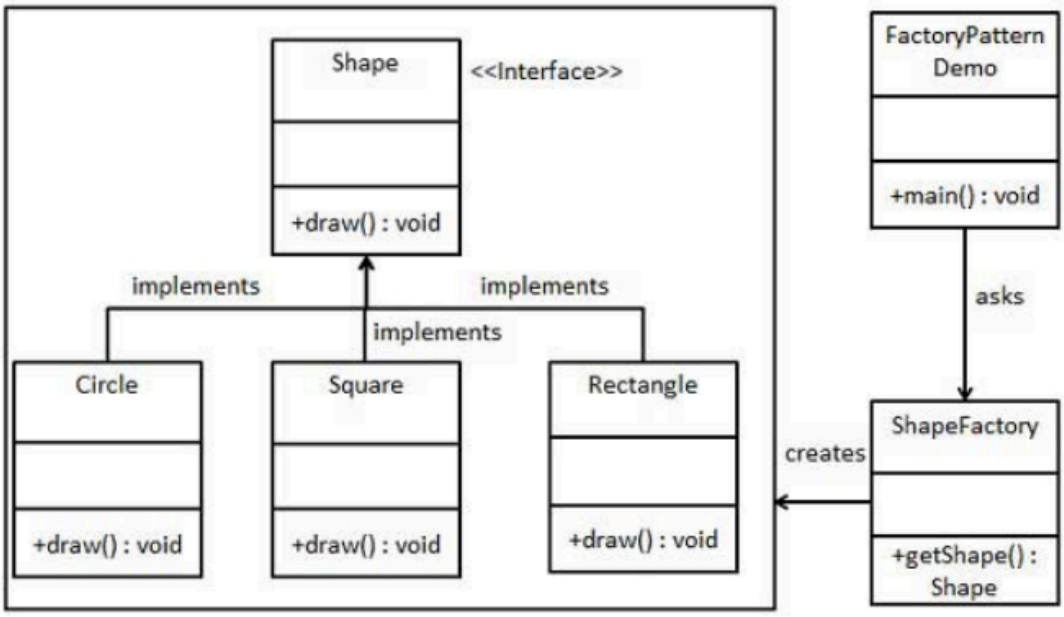
Disadvantages

- Increased complexity: Can add a layer of abstraction.
- Overkill for simple cases: Might not be necessary if you have few object types with straightforward creation.

When to Use the Factory Pattern:

- Unpredictable object types: When you don't know the specific object types you need until runtime (e.g., user selection or configuration files).
- Complex object creation: If creating objects involves complicated logic or special setups.
- Extensibility: When you need a system that's easily extended with new object types without constantly changing existing code.

- Object lifecycle management: If you need special handling of object creation or reuse.



Abstract Factory Pattern:

The Abstract Factory Pattern builds upon the Factory Pattern. It's like having a group of factories, each specializing in producing families of related objects.

The Abstract Factory Design Pattern is a creational design pattern that provides an interface for creating families of related objects without specifying their concrete classes. It is an extension of the Factory Pattern and is **used when there are multiple related objects to be created**.

Purpose

- Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- Encapsulates the management of a set of concrete factories.

Structure

- **AbstractFactory:**
    - Declares an interface for creating a set of abstract products.
- **ConcreteFactory:**
    - Implements the operations from the AbstractFactory to create concrete product objects. There's a ConcreteFactory for each product family.
- **AbstractProduct**
    - Defines an interface for a type of product object.
- **ConcreteProduct:**
    - Implements the AbstractProduct interface, representing specific product implementations within a family.

- **Client:**
    - Interacts only with the AbstractFactory and AbstractProduct interfaces.

When to Consider the Abstract Factory Pattern

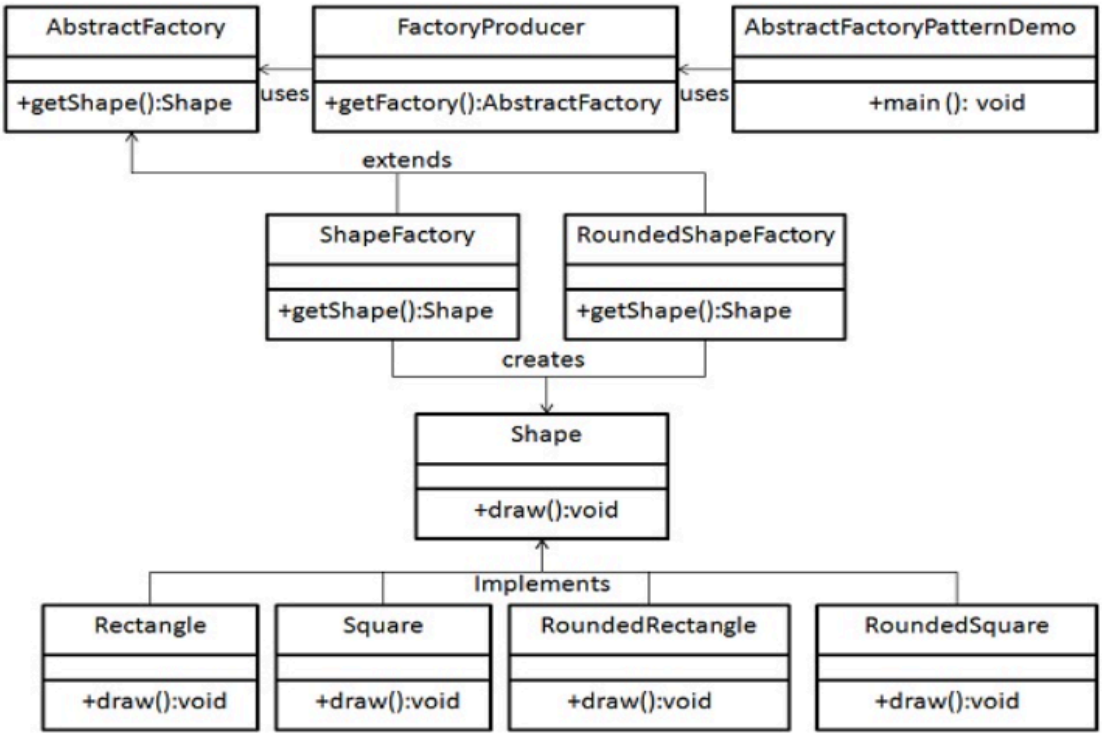
- **You have multiple families of related products that need to be used together.**
- You want to ensure that clients only create products within a compatible family and not mixed between families.
- **You want to provide a library of object families without exposing the concrete classes that implement them.**

Advantages

- **Encapsulation:** Hides concrete classes that implement product families, making the system more flexible.
- **Interchangeability:** You can swap out entire families of objects at runtime.
- **Open/Closed Principle:** You can introduce new product families without breaking existing client code.

Disadvantages

- **Complexity:** Can be more complex than a simple Factory Pattern.


AbstractFactory
+getShape():Shape
uses
FactoryProducer
+getFactory():AbstractFactory
uses
AbstractFactoryPatternDemo
+main (): void
extends
ShapeFactory
+getShape():Shape
RoundedShapeFactory
+getShape():Shape
creates
Shape
+draw():void
Implements
Rectangle
+draw():void
Square
+draw():void
RoundedRectangle
+draw():void
RoundedSquare
+draw():void

Singleton:

The Singleton design pattern is a creational pattern that ensures only a single instance of a class can ever exist while providing a global point of access to that instance.

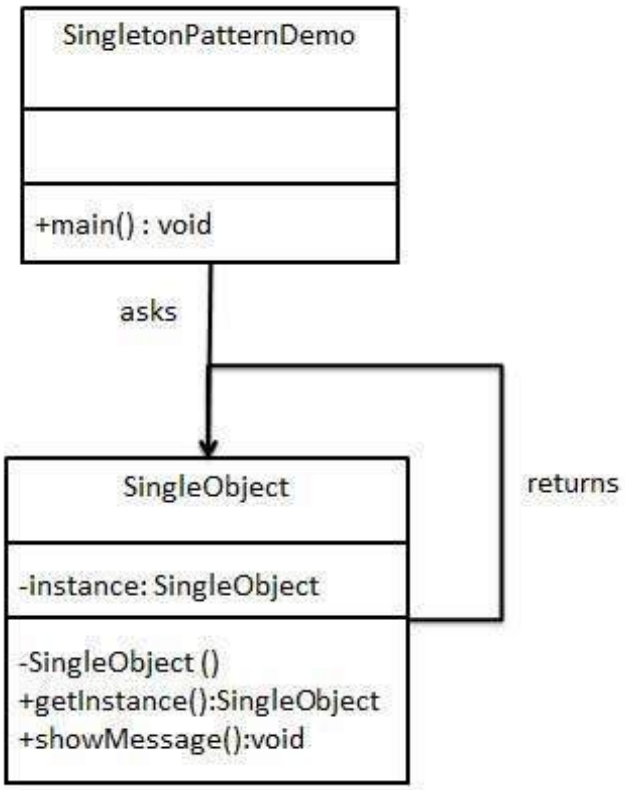
Singleton classes are used for logging, driver objects, caching, and thread pool, database connections.

Why Singleton?

• Application needs one, and only one, instance of an object!

• Sometimes we need to have only one instance of our class for example a single DB connection shared by multiple objects as creating a separate DB connection for every object may be costly.

• Similarly, there can be a single configuratipn manager or error manager in an application that handles all problems instead of creating multiple managers.



Pros

Ensure a class only has one instance

Provide a global point of access to it.

Controlled access to a sole instance

Reduced namespace

Permits refinement of operations and representation

Permits a variable number of instances

Cons:

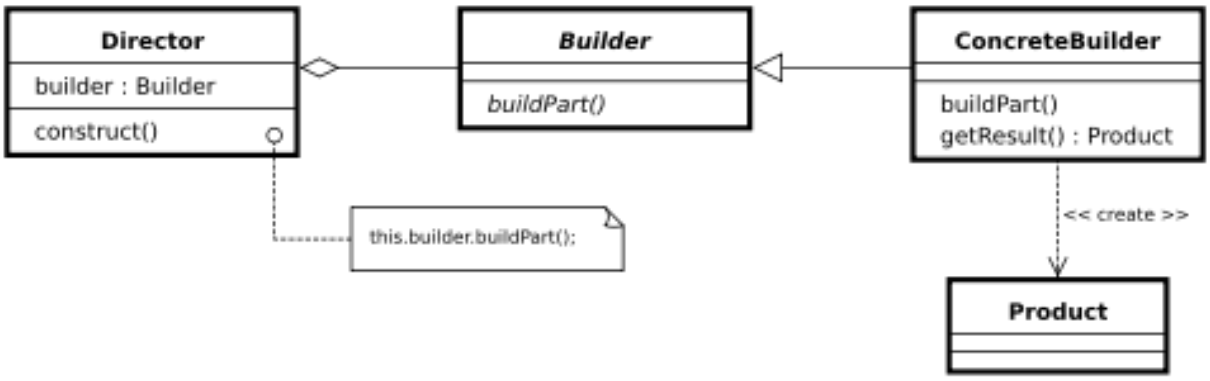
Because Singletons introduce a global state, unit testing can become challenging. Testing one component in isolation may be more complicated if it relies on a Singleton,

Builder Pattern:

The Builder Pattern is a design pattern used in software development to construct complex objects step by step. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Common Use Cases for the Builder Pattern:
Complex Objects: Builder pattern simplifies creating complex objects with many options or steps.
Immutable Objects: The builder pattern is ideal for immutable objects, promoting safety and predictability.
Flexible Object Initialization:
Fluent Interfaces (Optional): Builders often use method chaining for a clean, step-by-step construction process.

Flexibility: Easy to add new object variations with the Builder pattern.

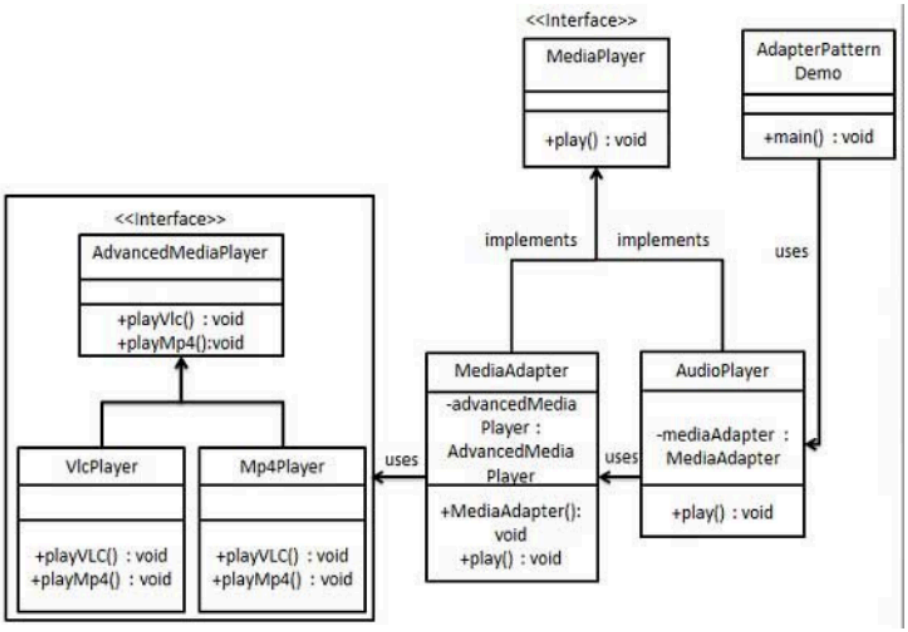


Aspect	Builder Pattern	Abstract Factory Pattern
Intent	Separates the construction of a complex object from its representation, allowing the same construction process to create multiple representations.	Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
Level of Granularity	Fine-grained control over individual object construction, allowing customization of attributes or configurations.	Higher level of abstraction, creating entire families of related objects without specifying their concrete implementations.
Object Construction	Constructs individual complex objects step by step, with a fluent interface for setting attributes or configurations.	Creates families of related objects, where each factory produces objects with compatible interfaces but different implementations.
Relationship	Focuses on building a single complex object with many configuration options.	Focuses on creating families of related objects with consistent interfaces but different implementations.
Example	Creating a User object with various optional attributes using a UserBuilder.	Producing different UI components such as buttons, checkboxes, and text fields using a GUIFactory.

Adapter: Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

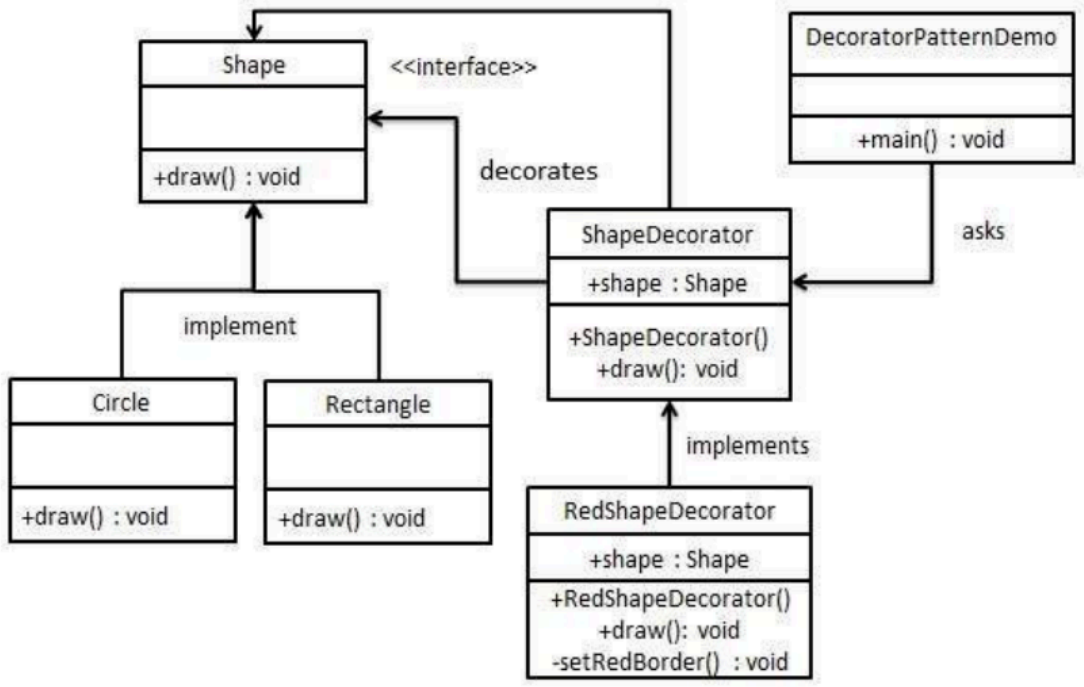
The Adapter pattern is commonly used in scenarios where there's a need to bridge the gap between two incompatible interfaces. Here are some concise use cases:

1. Legacy System Integration
2. API Versioning
3. Database Connectivity
4. Third-Party Library Integration
5. Device Connectivity
6. Logging and Monitoring



Design Patterns - Decorator Pattern

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to an existing class. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.



Observer Design Pattern

The observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

It is also referred to as the publish-subscribe pattern. In observer pattern, there are many observers (subscriber objects) that are observing a particular subject (publisher object). Observers register themselves to a subject to get a notification when there is a change made inside that subject. A observer object can register or unregister from subject at any point of time. It helps is making the objects objects loosely coupled.

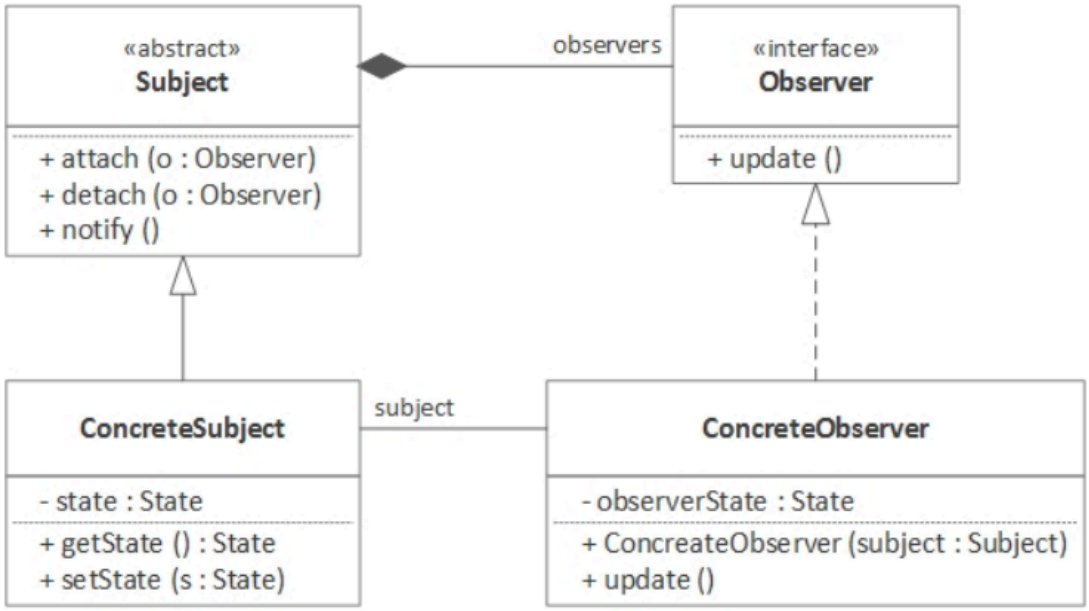
Pros:

1. Loose coupling between objects.
2. Flexibility and extensibility.
3. Easier maintenance and modification.
4. Reusability of components.
5. Supports event-driven architecture.

Cons:

1. Potential for unexpected updates.
2. Order dependency of observers.
3. Risk of memory leaks if not properly managed.
4. Debugging complexity may increase.
5. Adds complexity to the codebase.

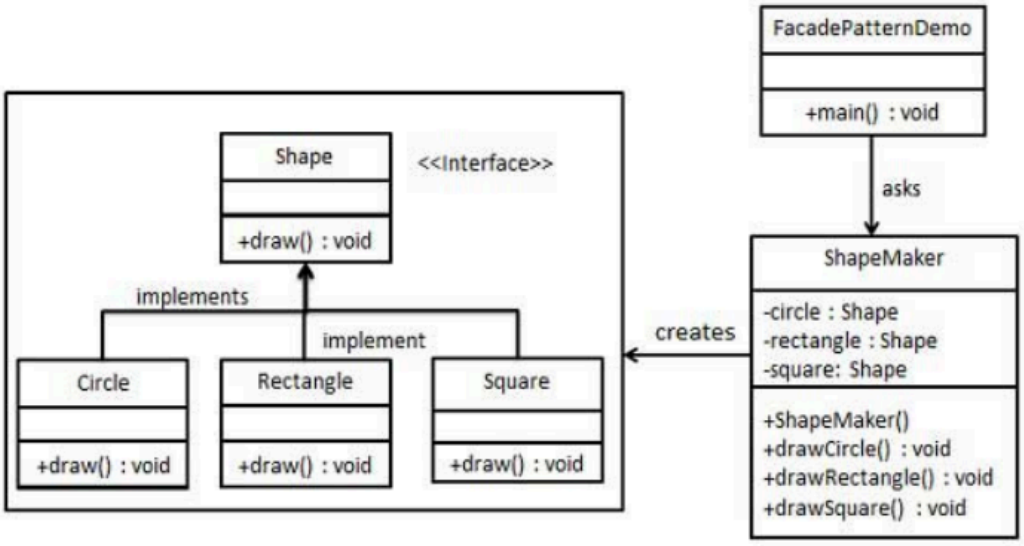
3.1. Architecture



Observer Pattern Architecture

Facade Pattern

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to existing system to hide its complexities. This pattern involves a single class which provides simplified methods required by client and delegates calls to methods of existing system classes.


FacadePatternDemo
+main() : void
asks
ShapeMaker
-circle : Shape
-rectangle : Shape
-square: Shape
+ShapeMaker()
+drawCircle() : void
+drawRectangle() : void
+drawSquare() : void
creates
Shape
<<Interface>>
+draw() : void
implements
implement
Circle
+draw() : void
Rectangle
+draw() : void
Square
+draw() : void