Introduction to Design pattern

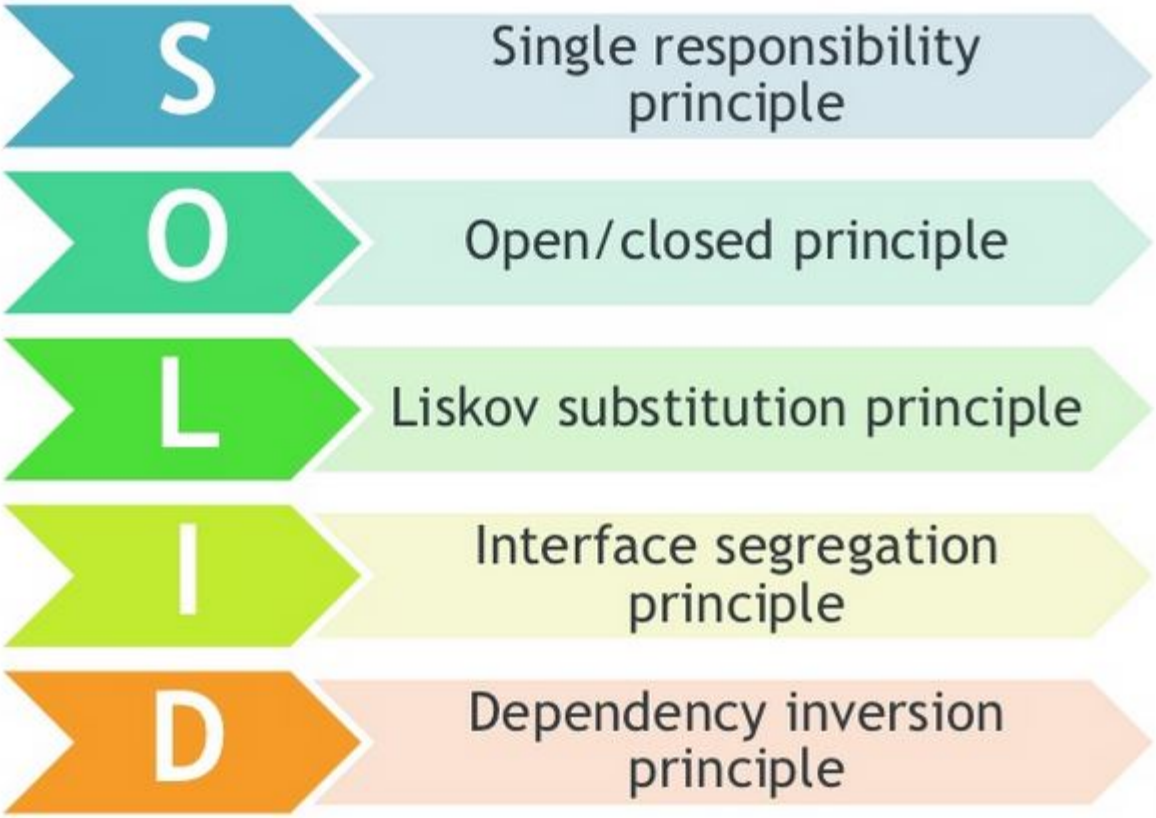
Rajeev Gupta
Java trainer
Rgupta.mtech@gmail.com

Design pattern

- Proven way of doing things
- **Gang of 4 design patterns ???**
- **total 23 patterns**
- **Classification patterns**
  1. **Creational**
  2. **Structural**
  3. **Behavioral**

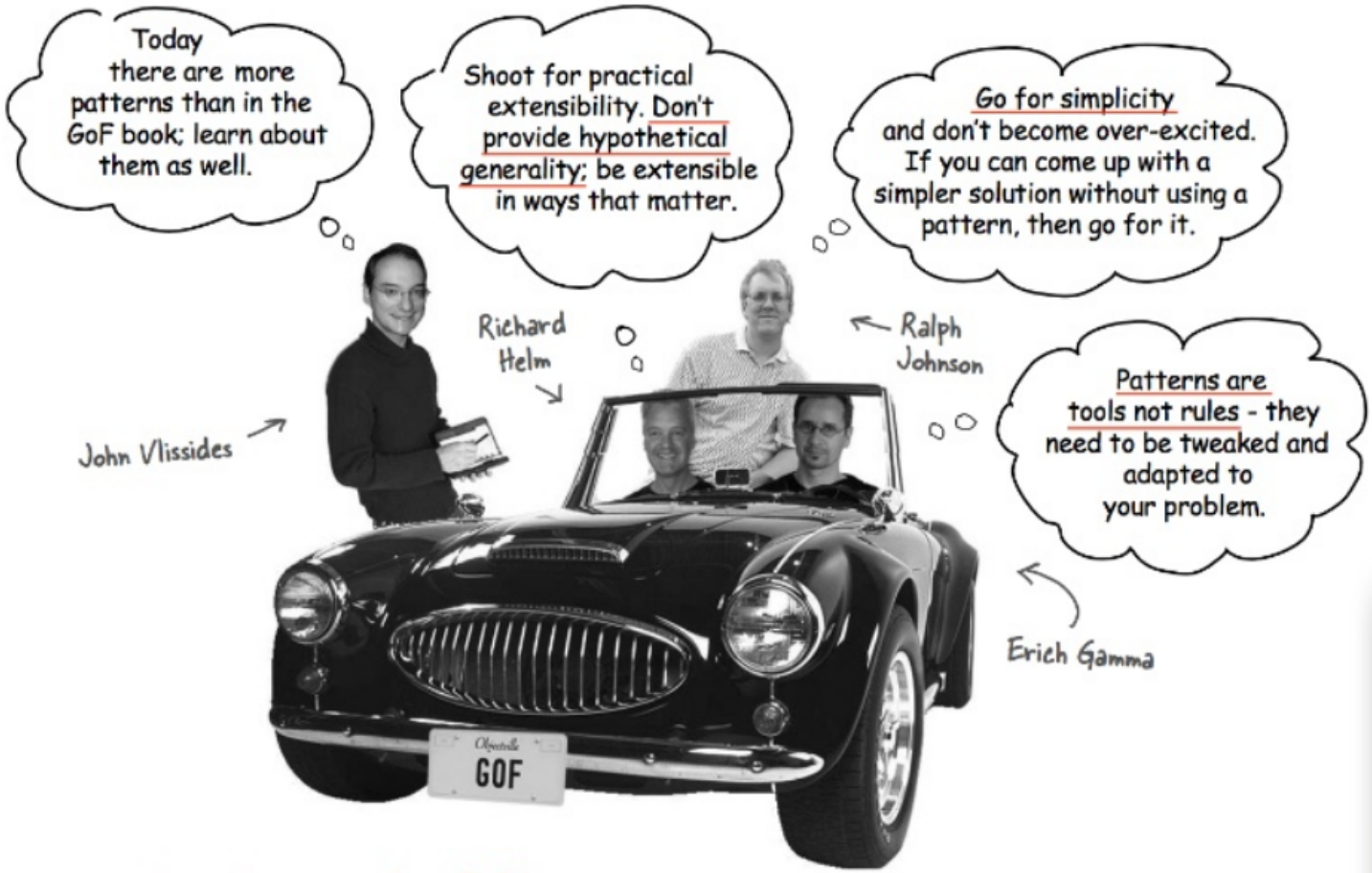
Top 10 Object Oriented Design Principles

1. DRY (Don't repeat yourself) - avoids duplication in code.
2. Encapsulate what changes - hides implementation detail, helps in maintenance
3. Open Closed design principle - open for extension, closed for modification
4. SRP (Single Responsibility Principle) - one class should do one thing and do it well
5. DIP (Dependency Inversion Principle) - don't ask, let framework give to you
6. Favor Composition over Inheritance - Code reuse without cost of inflexibility
7. LSP (Liskov Substitution Principle) - Sub type must be substitutable for super type
8. ISP (Interface Segregation Principle) - Avoid monolithic interface, reduce pain on client side
9. Programming for Interface - Helps in maintenance, improves flexibility
10. Delegation principle - Don't do all things by yourself, delegate it


S
Single responsibility principle
O
Open/closed principle
L
Liskov substitution principle
I
Interface segregation principle
D
Dependency inversion principle

"In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. It is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations."

WIKIPEDIA
The Free Encyclopedia



Keep it simple (KISS)

Pattern

- **Context**
  - a design situation giving rise to a design problem
- **Problem**
  - a set of **forces** occuring in that context
- **Solution**
  - a form or rule that can be applied to **resolve** these forces

IF you find yourself in **CONTEXT**
for example **EXAMPLES**,
with **PROBLEM**,
entailing **FORCES**
THEN for some **REASONS**,
apply **DESIGN FORM AND/OR RULE**
to construct **SOLUTION**
leading to **NEW CONTEXT & OTHER PATTERNS**

Design Patterns – Classification

Structural Patterns	Creational Patterns	Behavioral Patterns
• 1. Decorator • 2. Proxy • 3. Bridge • 4. Composite • 5. Flyweight • 6. Adapter • 7. Facade	• 1. Prototype • 2. Factory Method • 3. Singleton • 4. Abstract Factory • 5. Builder	• 1. Strategy • 2. State • 3. TemplateMethod • 4. Chain of Responsibility • 5. Command • 6. Iterator • 7. Mediator • 8. Observer • 9. Visitor • 10. Interpreter • 11. Memento

Patterns classification

- Creational patterns?
  - What is the best way to create object, new is not the best option
- structural patterns
  - Structural Patterns describe how objects and classes can be combined to form larger structures
- Behavioral Patterns
  - Behavioral patterns are those which are concerned with interactions between the objects (talking to each other still loosely coupled)

Creational Patterns

Creational Patterns

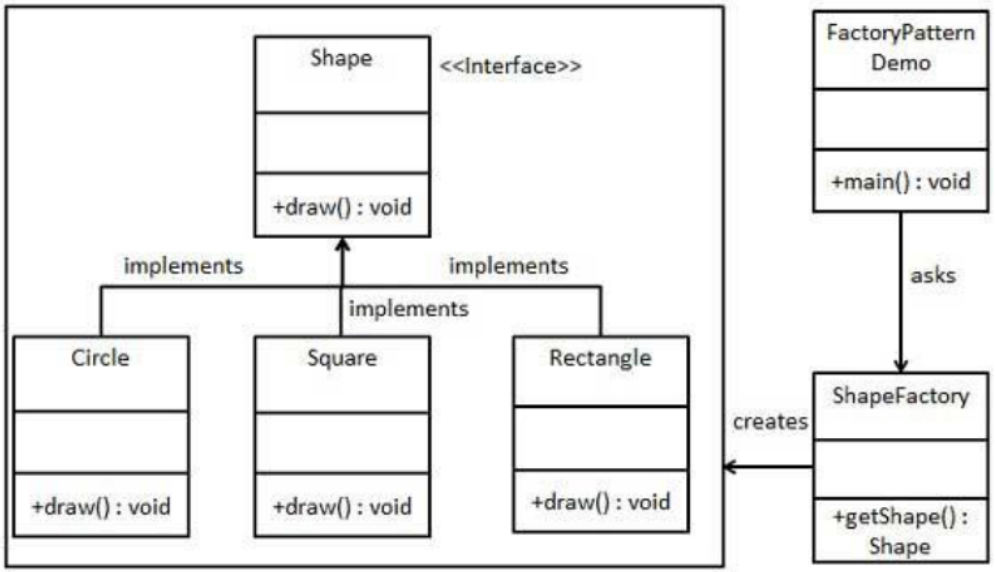
Book book = new Book ();

All the Creational patterns define the best possible way in which an object can be instantiated.

- The new Operator creates the instance of an object, but this is hard-coding.

- we can make use of Creational Patterns to give this a more general and flexible approach.

Simple Factory



- **Factory of what? of classes. ..**

In simple words,

“if we have a super class and an sub-classes, and based on data provided, we have to return the object of one of the sub-classes, we use a simple factory”

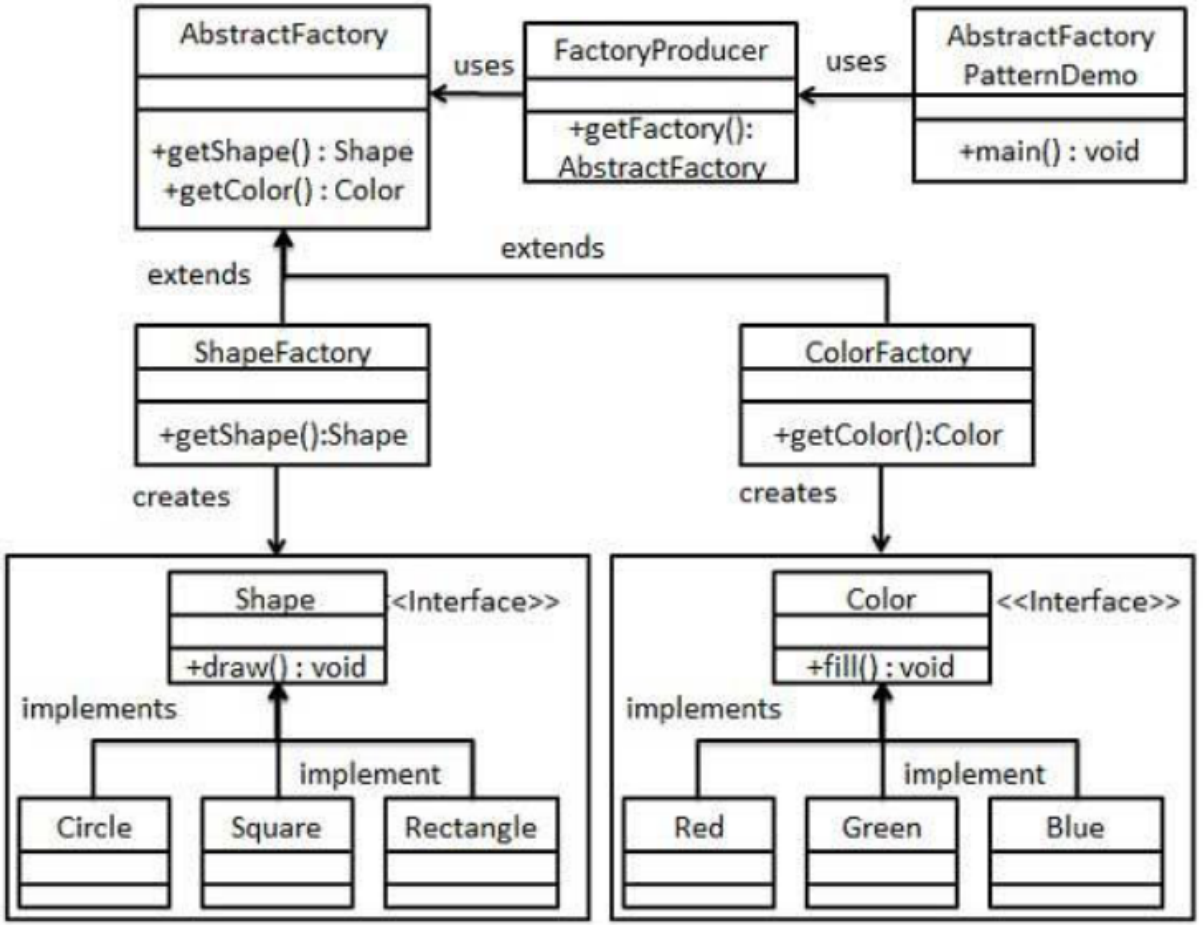
Factory Method Design Pattern

Define an **interface** for creating an object, but let **subclasses** decide which object to **instantiate**. Factory Method lets a class defer instantiation to subclasses.

Abstract Factory Pattern

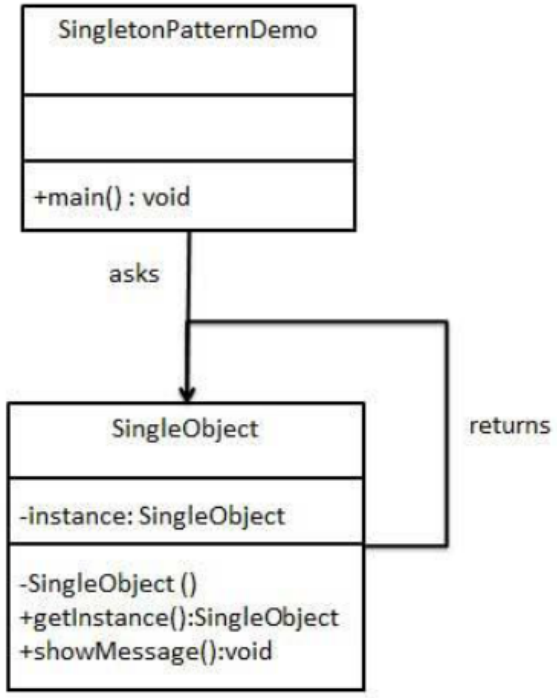
- This pattern is one level of abstraction higher than factory pattern. This means that the abstract factory returns the factory of classes
- Like Factory pattern returned one of the several sub-classes, this returns such factory which later will return one of the sub-classes.

Abstract Factory Pattern



Singleton Design Pattern

- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.
- The singleton class must provide a global access point to get the instance of the class.
- 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.



Singleton Design Consideration

Eager initialization
Static block initialization
Lazy Initialization
Thread Safe Singleton
Serialization issue
Cloning issue
Using Reflection to destroy Singleton Pattern
Enum Singleton
Best programming practices

Builder Pattern

- The Builder pattern can be used to ease the construction of a complex object from simple objects.
- The Builder pattern also separates the construction of a complex object from its representation so that the same construction process can be used to create another composition of objects.
- Need of builder is more then that of factory pattern because we aren’t returning objects which are simple descendents of a base display object, but totally different user interfaces made up of different combinations of display objects
- Separates object construction from its representation
- Builder pattern is useful when the construction of the object is very complex.
- The main objective is to separate the construction of objects and their representations. If we are able to separate the construction and representation, we can then get many representations from the same construction.
-

Prototype Pattern

- The prototype means making a clone.
- This implies cloning of an object to avoid creation. If the cost of creating a new object is large and creation is resource intensive, we clone the object.
- We use the interface Cloneable and call its method clone() to clone the object.

Structural Patterns

Adapter Pattern

The Adapter pattern is used so that two unrelated interfaces can work together.

When you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

- A class adapter
  - uses multiple inheritance (by extending one class and/or implementing one or more classes) to adapt one interface to another.
- An object adapter
  - relies on object aggregation.

Decorator design pattern

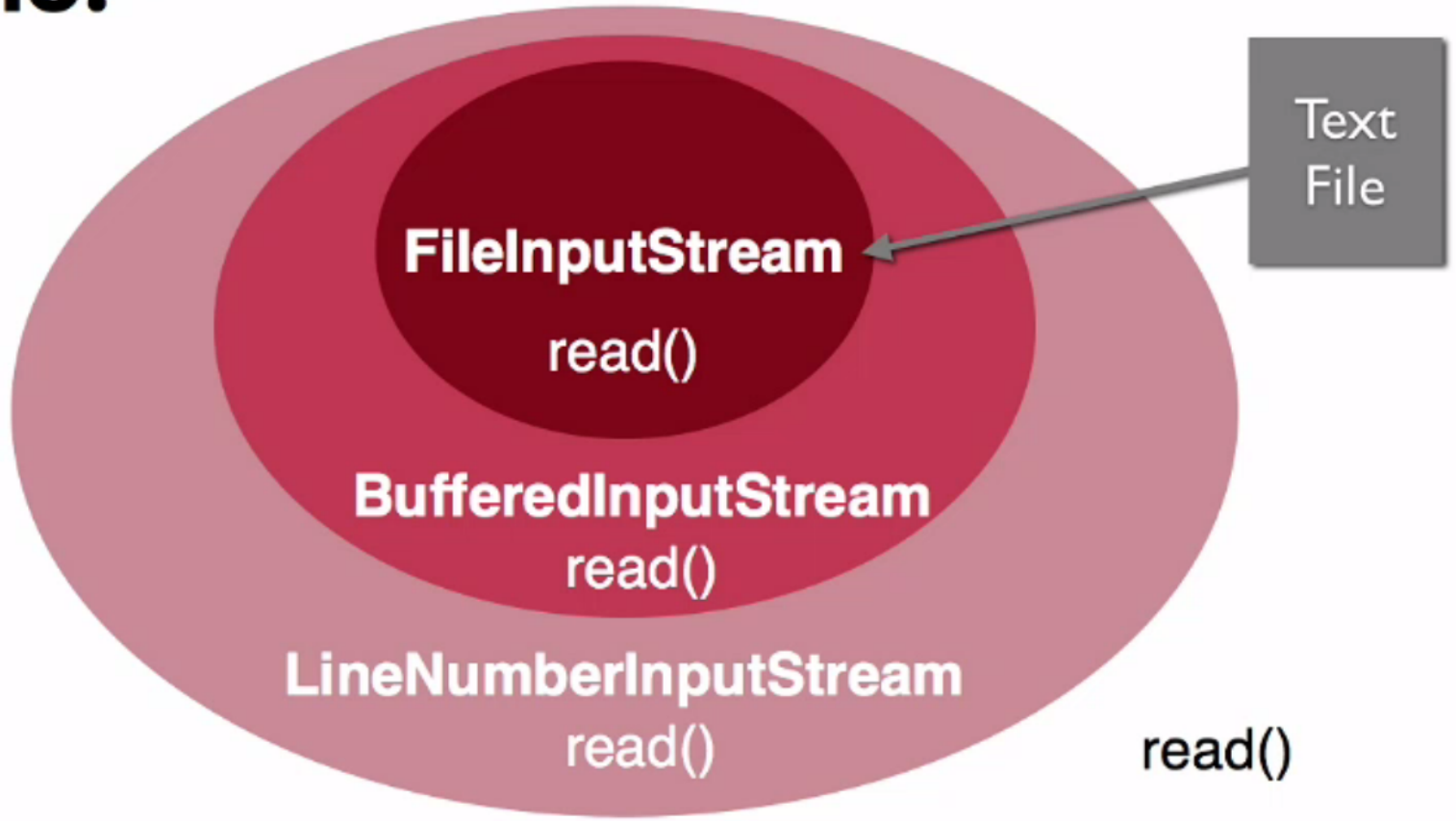
- Series of wrapper class that define functionality
- In the Decorator pattern, a decorator object is wrapped around the original object.
- The decorator must conform to the interface of the original object (the object being decorated).

Adding behaviour statically or dynamically

Extending functionality without effecting the behaviour of other objects.

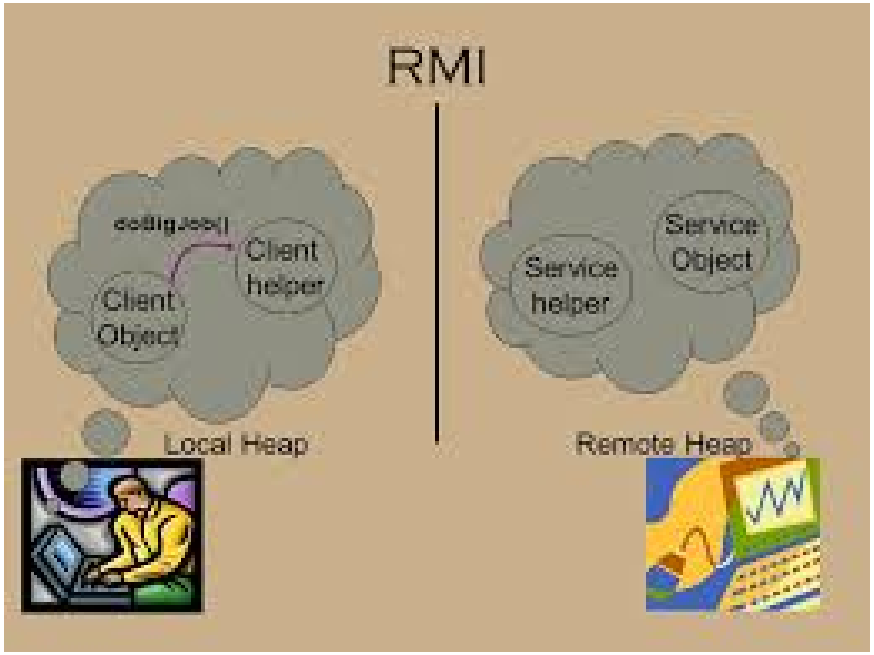
Adhering to Open for extension, closed for modification.

java.io.*


Text
File
FileInputStream
read()
BufferedInputStream
read()
LineNumberInputStream
read()
read()

Proxy design pattern

Proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern. In short, a proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes.



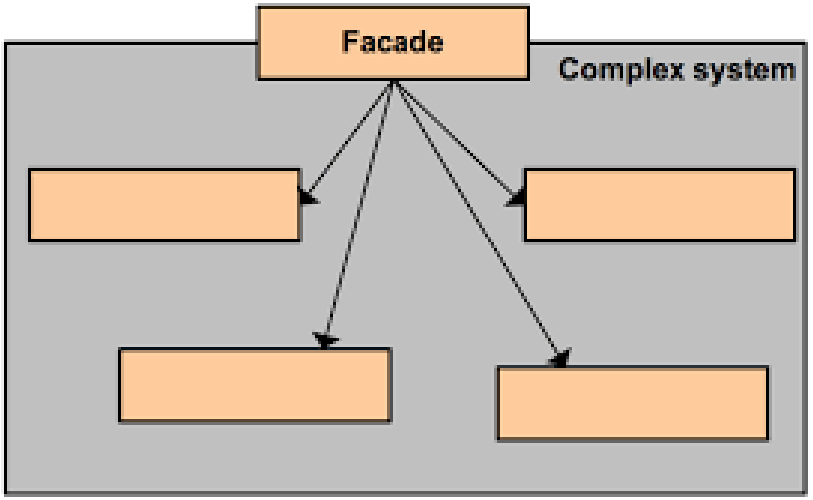
Facade pattern

The **facade pattern** (also spelled *façade*) is a software design pattern commonly used with object-oriented programming. The name is an analogy to an architectural façade.

A facade is an object that provides a simplified interface to a larger body of code, such as a class library. A facade can

- make a software library easier to use, understand, and test, since the facade has convenient methods for common tasks,
- make the library more readable, for the same reason,
- reduce dependencies of outside code on the inner workings of a library, since most code uses the facade, thus allowing more flexibility in developing the system,
- wrap a poorly designed collection of APIs with a single well-designed API.

The Facade design pattern is often used when a system is very complex or difficult to understand because the system has a large number of interdependent classes or its source code is unavailable. This pattern hides the complexities of the larger system and provides a simpler interface to the client. It typically involves a single wrapper class that contains a set of members required by the client. These members access the system on behalf of the facade client and hide the implementation details.



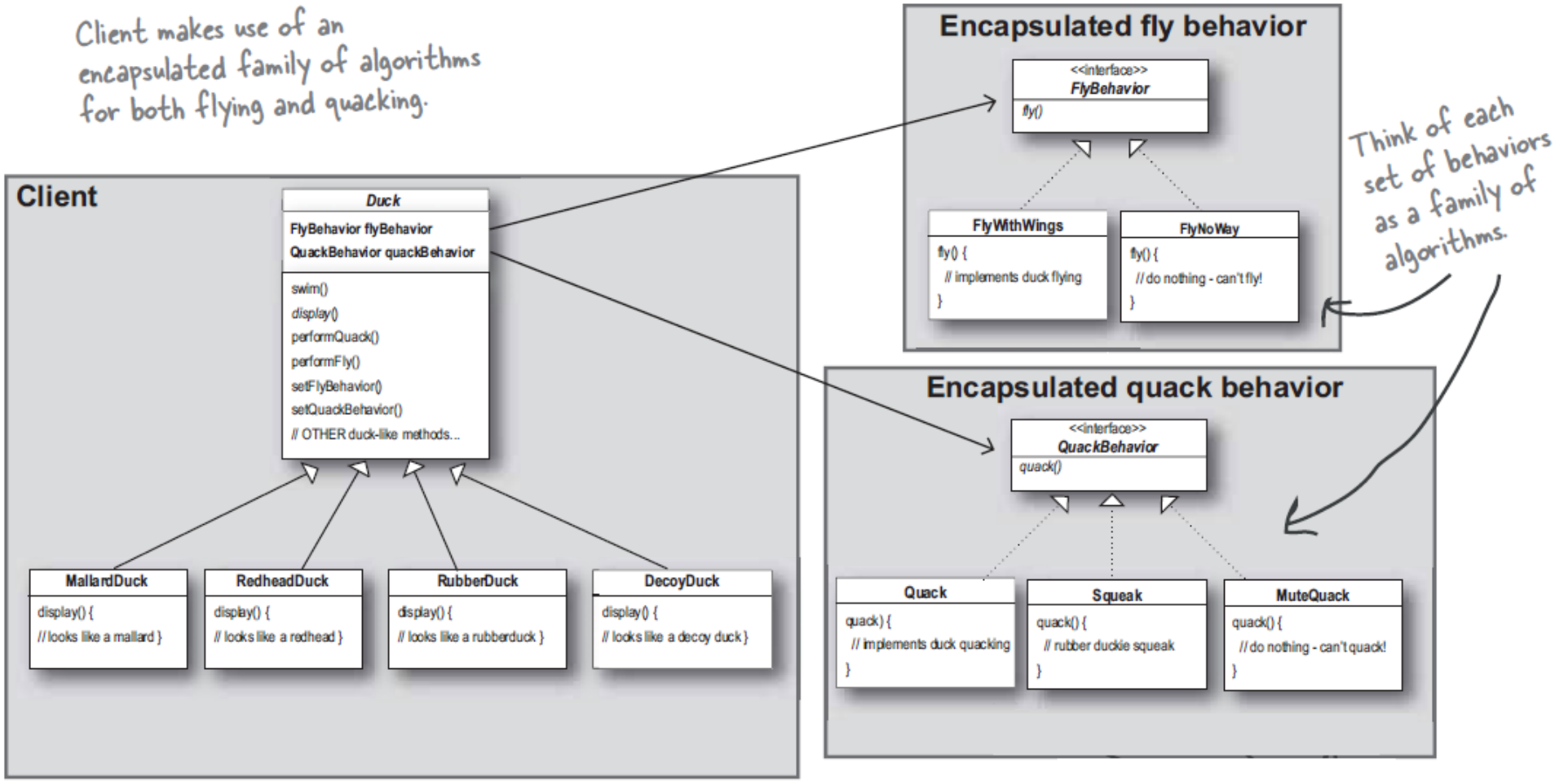
Behavioural Patterns

Strategy pattern /policy pattern

- The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable

- It is useful for situations where it is necessary to dynamically swap the algorithms used in an application.

- select algo at run time
- convert IS-A to HAS-A


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}

Iterator Design Pattern

- The iterator pattern encapsulates iteration.
- The iterator pattern requires an interface called *Iterator*.
- The *Iterator* interface has two methods:
  - *hasNext()*
  - *next()*
- Iterators for different types of data structures are implemented from this interface.