



Design Patterns(SE-408)

Course Teacher

Assistant Professor

Engr. Mustafa Latif



1



Introduction to Design Patterns

2



What's a Design Pattern?

- **Design patterns** are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.
- You can't just find a pattern and copy it into your program, the way you can with [off-the-shelf functions or libraries](#). The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program.

3



Algorithms vs. Pattern

- An algorithm always defines a clear set of actions that can achieve some goal, a pattern is a more high-level description of a solution. The code of the same pattern applied to two different programs may be different.
- An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal. On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

4



Who invented patterns?

- The concept of patterns was first described by **Christopher Alexander** in *“A Pattern Language: Towns, Buildings, Construction”*.
- The idea was picked up by four authors: **Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm**. In 1995, they published *“Design Patterns: Elements of Reusable Object-Oriented Software”*, also known as “the GOF book” in which they applied the concept of design patterns to programming.

5



Why Should I Learn Patterns?

- Design patterns are a toolkit of **tried and tested solutions** to common problems in software design. Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.
- Design patterns define a **common language** that you and your teammates can use to communicate more efficiently. You can say, “Oh, just use a Singleton for that,” and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern and its name.

6



What does the pattern consist of?

- **Intent** of the pattern briefly describes both the problem and the solution.
- **Motivation** further explains the problem and the solution the pattern makes possible.
- **Structure** of classes shows each part of the pattern and how they are related.
- **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern.

7



Classification of patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

8

Creational Design Patterns

9

Creational Design Patterns

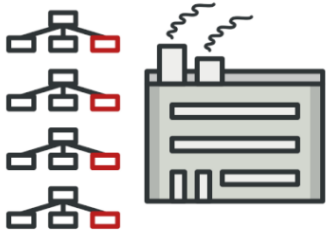


Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

10

Creational Design Patterns



Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

11

Creational Design Patterns

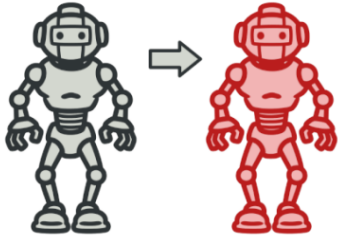


Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

12

Creational Design Patterns



Prototype

Lets you copy existing objects without making your code dependent on their classes.

13

Creational Design Patterns



Singleton

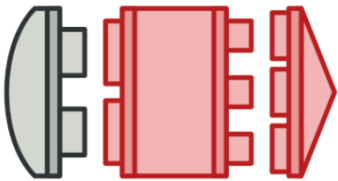
Lets you ensure that a class has only one instance, while providing a global access point to this instance.

14

Structural Design Patterns

15

Structural Design Patterns



Adapter

Provides a unified interface that allows objects with incompatible interfaces to collaborate.

16

Structural Design Patterns



Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

17

Structural Design Patterns

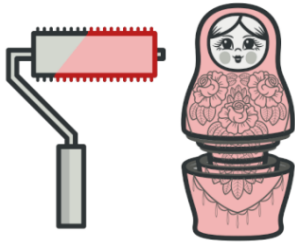


Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

18

Structural Design Patterns



Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

19

Structural Design Patterns

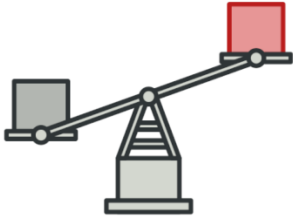


Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.

20

Structural Patterns

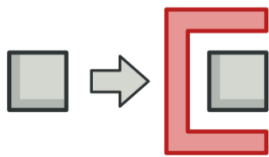


Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects, instead of keeping all of the data in each object.

21

Structural Design Patterns



Proxy

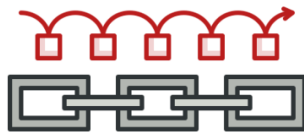
Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

22

Behavioral Design Patterns

23

Behavioral Design Patterns



Chain of Responsibility

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

24

Behavioral Design Patterns



Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.

25

Behavioral Design Patterns

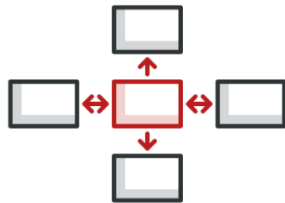


Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

26

Behavioral Design Patterns



Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

27

Behavioral Design Patterns

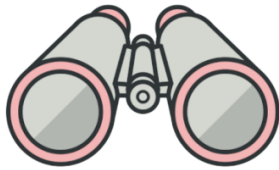


Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.

28

Behavioral Design Patterns

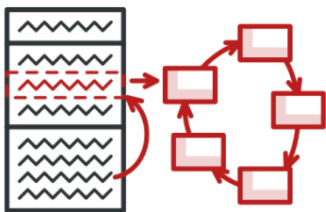


Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

29

Behavioral Design Patterns

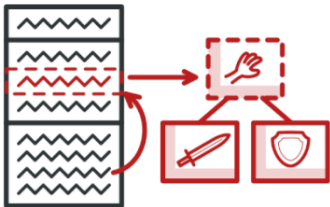


State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

30

Behavioral Design Patterns



Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

31

Behavioral Design Patterns



Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

32

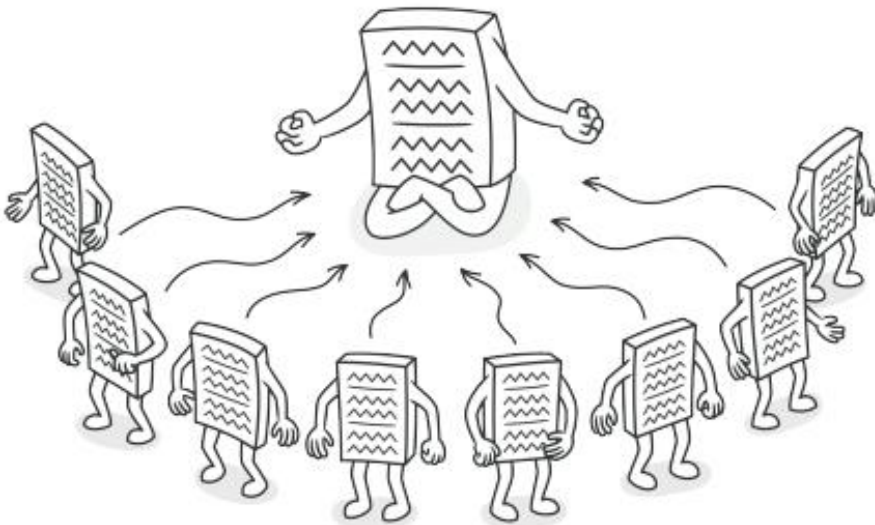
Behavioral Design Patterns



Visitor

Lets you separate algorithms from the objects on which they operate.

33



SINGLETON PATTERN

34

Singleton Pattern

- *Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.*

35

Problem

The Singleton pattern solves two problems at the same time, violating the Single Responsibility Principle:

Ensure that a class has just a single instance. Why would anyone want to control how many instances a class has? The most common reason for this is to control access to some shared resource—for example, a database or a file.

Here's how it works: imagine that you created an object, but after a while decided to create a new one. Instead of receiving a fresh object, you'll get the one you already created.

Note that this behavior is impossible to implement with a regular constructor since a constructor call must always return a new object by design.

36

Problem

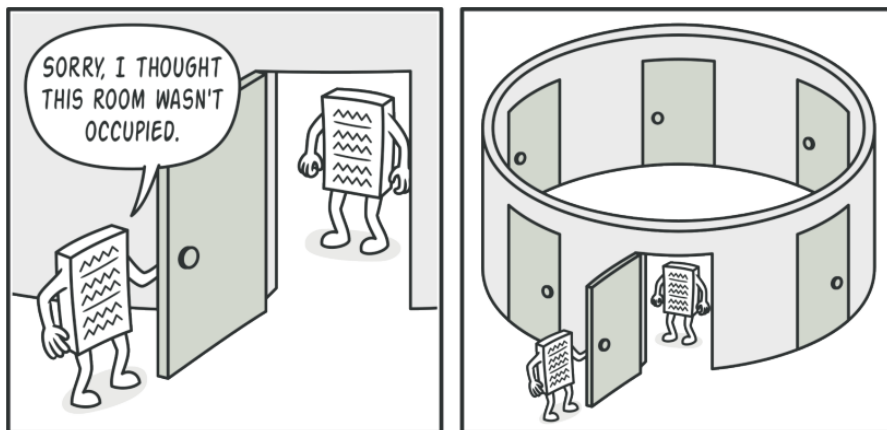
Provide a global access point to that instance. Remember those global variables that you (all right, me) used to store some essential objects? While they're very handy, they're also very unsafe since any code can potentially overwrite the contents of those variables and crash the app. Just like a global variable, the Singleton pattern lets you access some object from anywhere in the program. However, it also protects that instance from being overwritten by other code.

There's another side to this problem: you don't want the code that solves problem #1 to be scattered all over your program. It's much better to have it within one class, especially if the rest of your code already depends on it.

37

Problem

Nowadays, the Singleton pattern has become so popular that people may call something a singleton even if it solves just one of the listed problems.



Clients may not even realize that they're working with the same object all the time.

38

Solution

All implementations of the Singleton have these two steps in common:

- Make the default constructor private, to prevent other objects from using the new operator with the Singleton class.
- Create a static creation method that acts as a constructor . Under the hood, this method calls the private constructor to create an object and saves it in a static field. All following calls to this method return the cached object.

If your code has access to the Singleton class, then it's able to call the Singleton's static method. So whenever that method is called, the same object is always returned.

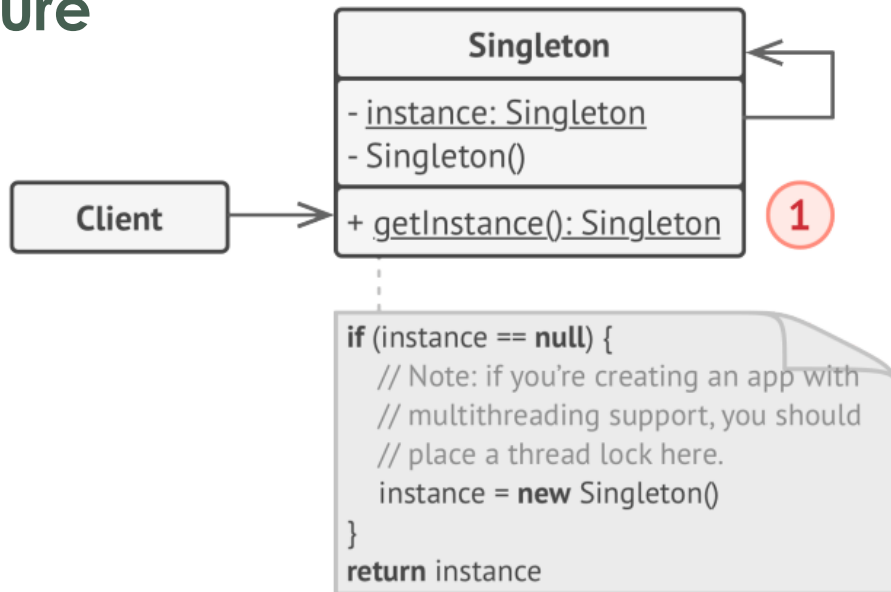
39

Real-World Analogy

- The government is an excellent example of the Singleton pattern. A country can have only one official government. Regardless of the personal identities of the individuals who form governments, the title, "The Government of X", is a global point of access that identifies the group of people in charge

40

Structure



41

1. The Singleton class declares the static method `getInstance` that returns the same instance of its own class.

The Singleton's constructor should be hidden from the client code. Calling the `getInstance` method should be the only way of getting the Singleton object.

42

Pseudocode


In this example, the database connection class acts as a Singleton.

This class doesn't have a public constructor, so the only way to get its object is to call the `getInstance` method. This method caches the first created object and returns it in all subsequent calls.

43


```
1 // The Database class defines the `getInstance` method that lets
2 // clients access the same instance of a database connection
3 // throughout the program.
4 class Database is
5   private field instance: Database
6
7 // The singleton's constructor should always be private to
8 // prevent direct construction calls with the `new`
9 // operator.
10 private constructor Database() is
11 // Some initialization code, such as the actual
12 // connection to a database server.
13 // ...
14
```

44




```
15 // The static method that controls access to the singleton
16 // instance.
17 static method getInstance() is
18 if (this.instance == null) then
19 acquireThreadLock() and then
20 // Ensure that the instance hasn't yet been
21 // initialized by another thread while this one
22 // has been waiting for the lock's release.
23 if (this.instance == null) then
24 this.instance = new Database()
25 return this.instance
```

45



```
26
27 // Finally, any singleton should define some business logic
28 // which can be executed on its instance.
29 public method query(sql) is
30 // For instance, all database queries of an app go
31 // through this method. Therefore, you can place
32 // throttling or caching logic here.
33 // ...
```

46



```
34
35 class Application is
36 method main() is
37 Database foo = Database.getInstance()
38 foo.query("SELECT ...")
39 // ...
40 Database bar = Database.getInstance()
41 bar.query("SELECT ...")
42 // The variable `bar` will contain the same object as
43 // the variable `foo`.
```

47



Java Code Example

48

Applicability

- Use the Singleton pattern when a class in your program should have just a single instance available to all clients; for example, a single database object shared by different parts of the program.
- Use the Singleton pattern when you need stricter control over global variables.

49

How to Implement

1. Add a private static field to the class for storing the singleton instance.
2. Declare a public static creation method for getting the singleton instance.
3. Implement “lazy initialization” inside the static method. It should create a new object on its first call and put it into the static field. The method should always return that instance on all subsequent calls.
4. Make the constructor of the class private. The static method of the class will still be able to call the constructor, but not the other objects.
5. Go over the client code and replace all direct calls to the singleton’s constructor with calls to its static creation method.

50



Pros and Cons

- You can be sure that a class has only a single instance.
- You gain a global access point to that instance.
- The singleton object is initialized only when it's requested for the first time.
- Violates the Single Responsibility Principle. The pattern solves two problems at the time.
- The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.