Design Patterns

Serban Scorteanu

Vlad Totea

Group 1100

Contents

[What are design patterns? 3](#_Toc93665197)

[Advantages and disadvantages 4](#_Toc93665198)

[Main Classification 5](#_Toc93665199)

[Creational Design Patterns 6](#_Toc93665200)

[ Factory 6](#_Toc93665201)

[ Singleton 11](#_Toc93665202)

[ Builder 14](#_Toc93665203)

[Structural Design Patterns 19](#_Toc93665204)

[ Decorator 19](#_Toc93665205)

[ Proxy Design Pattern 26](#_Toc93665206)

[Behavioural Design Patterns 32](#_Toc93665207)

[ Observer Design Pattern 32](#_Toc93665208)

[Conclusion 37](#_Toc93665209)

[Bibliography 38](#_Toc93665210)

# What are design patterns?

**Design patterns** = typical solutions to frequently occurring software design issues. They are not specific fragments of code, but instead they are concepts for solving specific problems

Usually, patterns are described in this manner:

**Intent** – this is where both the problem and the solution are described

**Motivation** – this is where the details of the problem and the solution are presented

**Structure** – the structure of the classes show how they are related and how they make up the pattern

**Example** – the actual code in order to prove the new solution and better understand the concept

# Advantages and disadvantages

**ADVANTAGES**

1. Reusability
2. Well-proved, well-tested solutions
3. Clarity and better system architecture
4. Better communication between developers

**DISADVANTAGES**

1. Complex
2. Seem to be simple but not really
3. Decrease understandability (by increasing code length and by using indirect flow – this indirection might actually reduce performance)
4. Can use more memory

# Main Classification

1. **Creational Design Patterns** (most frequently used)
2. Factory
3. Abstract Factory
4. Singleton
5. Prototype
6. Builder
7. **Structural Design Patterns**
8. Adapter
9. Bridge
10. Composite
11. Decorator
12. Facade
13. Flyweight
14. Proxy
15. **Behavioral Design Patterns**
16. Chain of Responsibility
17. Command
18. Interpreter
19. Iterator
20. Mediator
21. Memento
22. Observer
23. State
24. Strategy
25. Template
26. Visitor

# Creational Design Patterns

## Factory

The factory design pattern provides an interface for creating objects only in the parent class but gives permission to children classes to alter the type of objects which will be created.

In Factory – objects are created without exposing the logic to the user and the user will use the common interface to create new types of objects.

The implementation consists in creating a static factory method that will create and return instances.

Diagram

Description automatically generated

Best example of a problem and why this pattern was created:

We need to create a logistics application. The app was designed to handle truck transportation => most of the code is inside the Truck class. The app becomes so popular that sea transportation companies want to become our clients.

A picture containing diagram

Description automatically generated

Problem – most of the code is inside the Truck class, so adding Ships would require a lot of refactoring. Sometime after refactoring air transportation companies want to join.

Solution – replace direct object ctor calls with calls to a factory method which will be included in this method.

Diagram

Description automatically generated

Diagram, whiteboard

Description automatically generated

When should we use the Factory Pattern?

* When an abstract class / interface – expected to change frequently
* When we expect future changes and modify our current implementation would be tedious
* Text

  Description automatically generatedWhen the initialization is simple and the ctor needs few params

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

The factory could have also had children and we could have overridden the createShape() method inside these children in order to have even more control.

Another personal favourite example from Head First Design Patterns:

Diagram

Description automatically generated

Diagram

Description automatically generated

## Singleton

A picture containing diagram

Description automatically generated

Design pattern which ensures that a class has only one instance and also provides global access to that instance.

Best use cases: when we have a shared resource – databases, files.

Remember SOLID principles? Well, the Singleton pattern violates the Single Responsibility Principle.

The idea is simple. Imagine we have to create an object and after a while we want to create a new one. Even though we think we get a new one, we actually get a reference to the original one. This can’t be done with a regular ctor.

Diagram

Description automatically generated

This pattern is so popular some frameworks use it by default. Ex: Spring (Java’s most popular framework) uses it by default for all its beans. (there’s an extra step needed in order to change the scope of the beans to others such as prototype).

How do we implement Singleton?

2 steps:

1. The default constructor must be private so that other objects can’t use instantiate the singleton class
2. We have to create a static method that acts as a constructor. Behind the scenes (for the user) this method will call that private ctor and it will save it in a static field. From this point on any call to this method will return a reference to that original object.

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

Conclusion: why would we use the Singleton Pattern? Well, it lets us access the same object from anywhere and most importantly -> it protects that object/instance from being overwritten somewhere.

CONS: when multithreading we need a thread lock. The biggest issue with this pattern – unit testing. Most test frameworks create mock objects using inheritance. Because the constructor is private and static methods generally can’t be overridden.

Text

Description automatically generated

## Builder

A design pattern that helps constructing large, complex objects step by step. This pattern helps us create different representations, types of an object using the same code.

Diagram

Description automatically generated

PROBLEM: We have some houses. Some houses have a garage, some don’t have a garage but have swimming pools and statues, others have gardens and statues, etc. We have multiple similar objects but using inheritance would still be problematic as they have a lot of fields => a constructor with a lot of parameters which are breaking the rules of Clean Code and also prone to bugs.Website

Description automatically generated with low confidence

Diagram

Description automatically generated

What does it propose? How do we do it?

Well, the idea is to extract the object construction part out of the class and move it into so-called builder objects.

Diagram

Description automatically generated

Diagram

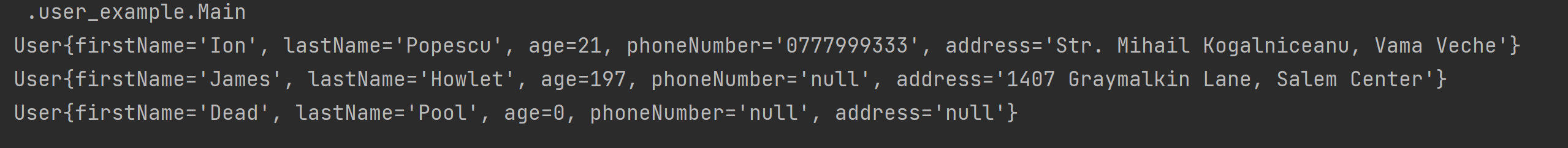
Description automatically generated

Text

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence



# Structural Design Patterns

Structural design patterns – focus on the relationships between classes – inheritance, as well as composition, tries to simplify the overall structure.

Their purpose – to simplify the design of large object structures, describing ways of composing so that classes and objects can become repeatable without refactoring each time.

The type of problem this category of patterns tries to solve – Example:

## Decorator

= ‘Wrapper’ pattern

= pattern that lets us attach new behaviours to objects by wrapping them using object which contain these behaviours

A picture containing linedrawing

Description automatically generated

A picture containing text, book

Description automatically generated

Example of problems which this pattern is meant to solve:

There’s an app needed – a notification library app – which will let other apps notify users about important events.

So, this app was created, and it will take care of sending emails about these events to the users.

After a while, the ‘other apps’ want to also send notification via SMS/Facebook/Slack. Ok, the developers of the notifier app will create three separate classes and inherit the notifier class (code is beginning to get large). OK, done. But someone comes out and says it – What if your house is on fire? I think you would want to get notified via everything – and this is where it gets messy with inheritance.

Instead of creating hundreds of subclasses, the decorator patterns reduce that subclass count by inserting “add-on classes”Diagram

Description automatically generated

Why is inheritance bad in this case? Well – inheritance is static and can’t be changed at runtime – you can only replace a whole object with another one. Also, child classes can only have one parent (at least in most programming languages)

Composition/Aggregation is the key here (and in most of the design patterns)

How do we implement it?

A person standing in front of a pile of boxes

Description automatically generated with low confidence

Implementation:

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

## Proxy Design Pattern

The proxy is a pattern which lets developers provide substitutes/placeholders for objects. It has control over the original object and allows actions before/after getting to the original object.

It basically allows the creation of an intermediary that will hide the complexity and act as an interface for the object.

Diagram

Description automatically generated

Example of problem solved by proxy:

We have a database in our application. As we know a database connection need a large number of resources, but we still need it so it’s not like we have a choice.

What if we need to execute the application sometimes without needing to query anything in the database? What if we need to access the database just before we close the app to save some data? Why would we slow the app by connecting to the db every time or connecting when we start the app instead of when we need it.

Text

Description automatically generated

We only want this kind of object to be initialized when we want, and once they are initialized, reuse them for all our calls.

A picture containing text

Description automatically generated

The main reasons for using proxy:

1. Need for a simplified version of a “heavy” resource/object – will load the og object on demand (lazy initialization instead of early) = **Virtual Proxy**
2. Need to represent locally an object at a different address space = **Remote Proxy**
3. Need to add security – to provide controlled access depending on the client’s permissions = **Protection Proxy**

Implementation:

Basic Structure – Interface inherited by both the original object and the proxy object. The client calls the interface, and he doesn’t know that this will actually call the proxy instead of the original object.

Diagram

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

A screen shot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated

# Behavioural Design Patterns

## Observer Design Pattern

Of all the behavioural design patterns, the Observer is the most used and best known.

It helps in defining a subscription mechanism in order to notify object about anything that happens to the observed object.

It specifies communication between entities: the **observable** and the **observers**. The observable is the object which notifies the others when it changes its state.

A picture containing engineering drawing

Description automatically generated

Problem which can be solved with Observer:

Two classes – Customer and Store. The customers are interested when to know when the new Iphone will become available.

One way – the customer goes every day to check the store.

Second way – the store will spam all the customers each time there’s a new product available.

Diagram

Description automatically generated

Solution:

The Observer will have two classes: Store and Subscriber (the previous Customer). The store will keep an array field for storing a list of references to subscribers and some public methods including the ones which will allow the subscribers to subscribe/unsubscribe. Now the store can spam only the subscribers. Everyone is happy.

Implementation:

Text

Description automatically generated

This will be equivalent to the Store class

Text

Description automatically generated

Text

Description automatically generated

This is the main class

Text

Description automatically generated

# Conclusion

In conclusion, we’ve presented the most common and frequently used patterns, but what you should remember from all this, is the fact that design patterns are typical solutions to common problems. They mostly have benefits as long as they are used where they should be used as each of them is meant to solve a specific design flaw.

Text

Description automatically generated

# Bibliography

Anon., n.d. *Design Patterns.* [Online]   
Available at: https://www.tutorialspoint.com/design\_pattern

Anon., n.d. *Java Core Patterns.* [Online]   
Available at: https://www.baeldung.com/java-core-patterns

Anon., n.d. *Java Design Patterns.* [Online]   
Available at: https://www.javatpoint.com/java-design-patterns

Freeman, E. & Robson, E., 2020. *Head First Design Patterns.* 2nd ed. s.l.:O'Reilly Media, Inc..

Guru, R., n.d. *Design Patterns.* [Online]   
Available at: https://refactoring.guru/design-patterns