Creational Design Patterns

(Singleton, Prototype)
using C++

Content

01- Introduction

- Definition and importance of design patterns in software design.
- Categories of design patterns.
- Purpose and benefits of creational patterns

02- Singleton Pattern

- Concept and definition
- A use case without using singleton pattern and its issues.
- Implementation steps
- Pros and Cons
- Real-world examples

03- Prototype Pattern

- Concept and definition
  - Explanation of prototype pattern
  - When to use prototype pattern
- Key characteristics
  - Cloning existing objects to create new ones
  - Avoids the cost of creating objects from scratch
- Implementation
  - Shallow vs. deep copy
- Pros and Cons
- Real-World examples

Content

04- Practical applications

- How to decide on using singleton or prototype in real-world scenarios
- Scenarios where singleton and prototype may be used together
- Guidelines for implementing these patterns effectively
- Common pitfalls and how to avoid them

05- Common pitfalls and how to avoid them

- Singleton pitfalls
- Prototype pitfalls
- Tips for robust implementation

Introduction:: Definition & importance in software design.

i. Definition:

Design patterns are general, reusable solutions to common problems that occur in software design. They aren't specific to a particular programming language but provide a template on how to solve a problem that can be used in many different situations.

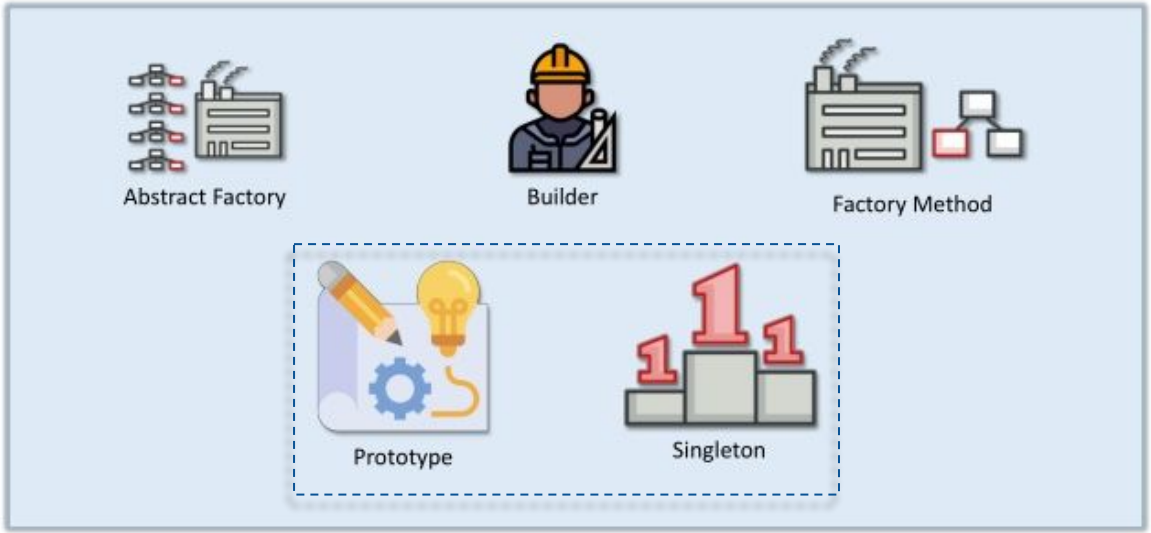
ii. Importance of design patterns in software design:

Efficiency	We can implement solutions faster because they don't need to solve common problems from scratch
Maintainability	Design patterns promote code that is easier to understand and maintain. Since these patterns are well-known, other developers can easily comprehend the design and purpose of the code
Scalability	Design patterns guide us in structuring the code in a way that it can grow and adapt to new requirements without significant rewrites.
Best practices	Design patterns encapsulates the best practices and principles of object oriented design, leading to better quality software.

Introduction:: Categories of design patterns

01- Creational Patterns:

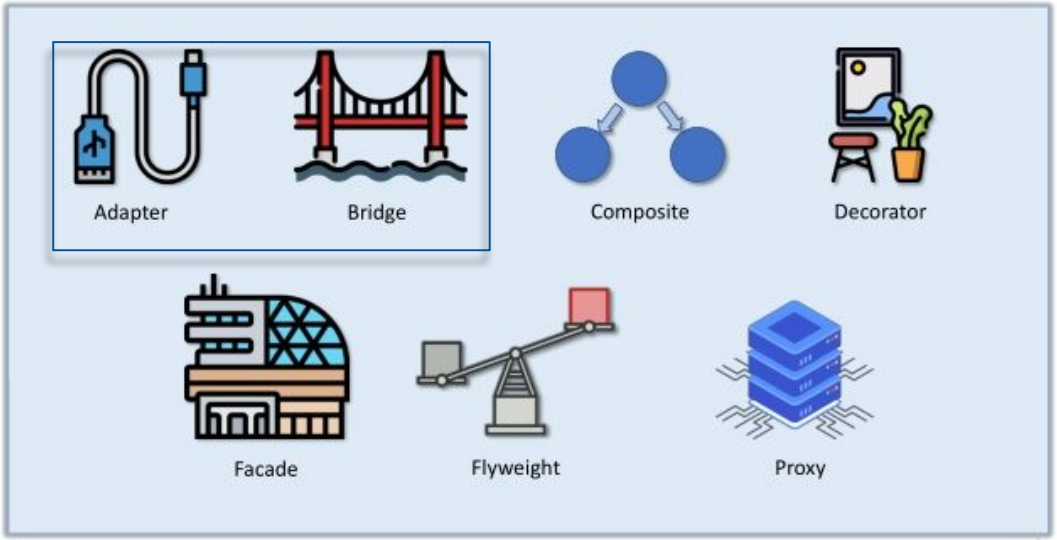
These patterns deal with object creation mechanisms, aiming to the situation. They help to make the system independent of how its objects are created.



Introduction:: Categories of design patterns

02- Structural Patterns:

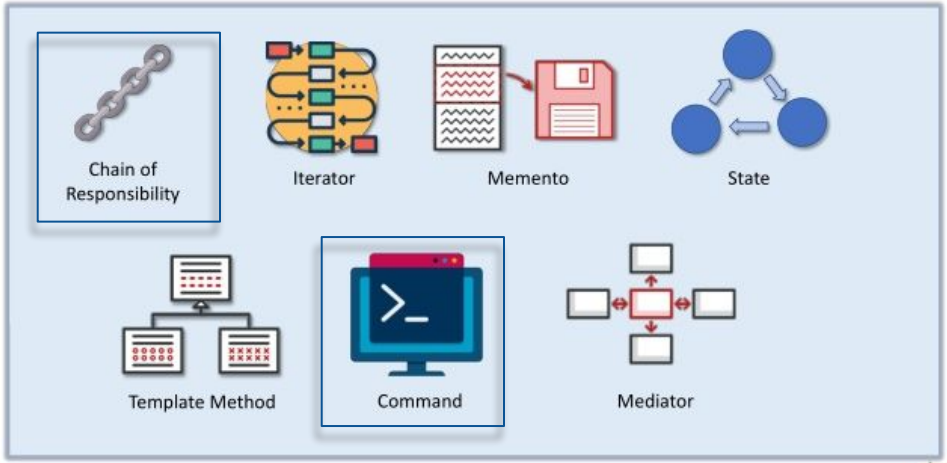
These patterns focus on how classes and objects are composed to form larger structures. They help ensure that if one part of a system changes, the entire structure doesn't need to be refactored.



Introduction:: Categories of design patterns

03- Behavioral Patterns:

These patterns are concerned with communication between objects, focusing on the flow of control and how responsibilities are assigned among objects.



Introduction:: Purpose and benefits of creational patterns

i. Purpose

Object creation control	Instead of creating objects directly, these patterns provide ways to create them that are more suited to different situations.
Separation of concerns	These patterns separate the process of creating objects from the code that uses those objects. This means that the code that uses the objects are created.
Flexibility in object creation	These patterns allows us to change the way objects are created without alerting the code uses the objects. This will make the code easier to be adapted to the requirements change,

Introduction:: Purpose and benefits of creational patterns

ii. Benefits after using the creational pattern in the code:

How these patterns change the code flexibility and Reusability?

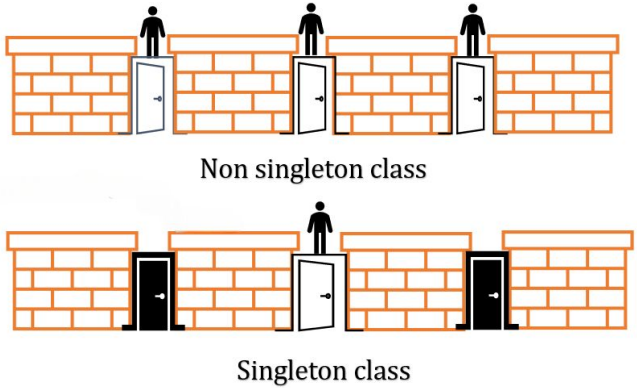
Improved flexibility	They help us generate an adaptable code, where we can easily change how objects are created based on the updated requirements.
Reusability	By using these patterns, the logic for creating objects will be centralized and can be reused across different parts of the program, achieving DRY concept (no code duplication).
Simplified code maintenance	Since creational patterns separate the object creation logic from the user/client/business logic, our code becomes easier to maintain without any effect on the rest of the program.

Singleton pattern:: Concept and definition

This is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. This pattern is particularly useful when one object is needed to coordinate actions across the system, such as a single database connection or a single configuration manager.

Singleton pattern ensures:

- A single instance of the class no matter how many times the class is accessed or how many threads are trying to create an instance. This is typically achieved by making the constructor of the class private and providing a static method to access the instance.
- It also provides a single, global point to access to the instance, which is through static method, allowing any part of the code to interact with the singleton object. This ensures that the same instance is used through the application.



Singleton pattern:: A use case without using singleton pattern and its issues.

Imagine you have an application that needs to read settings from a configuration file. The settings should be consistent throughout the whole application files, and there should only be one configuration manager to avoid conflicting data or unnecessary reloading the configuration. (This may happen if we created multiple objects of configuration manager class, each object will reload the configuration. The configurationManager class will be responsible for accessing the configuration from the configuration file).

Here is the approach without using Singleton pattern:

```
#include <iostream>
#include <string>

class ConfigurationManager {
public:
    ConfigurationManager() {
        std::cout << "Loading configuration from file...\n"; }
    std::string getConfigValue(const std::string& key) {
        return "Value for " + key;  }
};
```

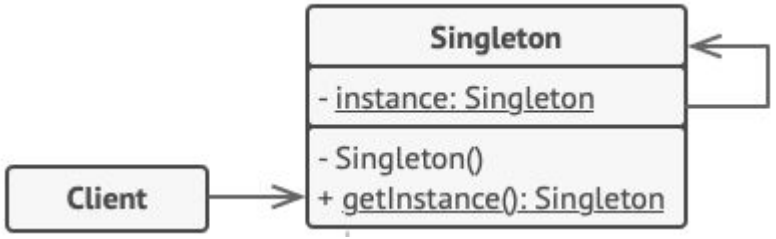
```
int main() {
    ConfigurationManager config1;  // First instance
    ConfigurationManager config2;  // Second instance

    std::cout << config1.getConfigValue("setting1") << std::endl;
    std::cout << config2.getConfigValue("setting2") << std::endl;

    return 0;
}
```

Issues in the previous approach:

- If we create multiple instances of the ConfigurationManager class (i.e. one instance per part of the application), each instance might load the configuration file separately which is inefficient. It also will be a waste for resources such as memory and will be time-consuming especially if the file is large, causing resource waste problem.
- Additionally, if one instance reads the file, and then the file is updated, another instance might read the updated file, leading to different parts of the application having different configurations. This problem is called inconsistent state.

Singleton pattern will help us solve these particular issues. So, How to implement it??

Here is the UML of Singleton pattern:



Singleton pattern:: Implementation steps

1. Private constructor: To prevent direct instantiation from outside the class.
2. Static method: To create and provide access to the Singleton instance. If instance doesn't exist yet, this method creates it; otherwise, it returns the existing instance.
3. Static variable: To hold the instance of the class, ensuring that it persists(shared) across multiple calls to `getInstance()`.

```
class ConfigurationManager {
private:
    static ConfigurationManager* instance;
    ConfigurationManager() {
        std::cout << "Loading configuration from file...\n";  }
public:
    static ConfigurationManager* getInstance() {
        if(instance == nullptr)
            instance = new ConfigurationManager();
        return instance;  }
    std::string getConfigValue(const std::string& key) {
        return "Value for " + key ;  }
};
```

```
ConfigurationManager* ConfigurationManager::instance =
nullptr;
int main() {
    ConfigurationManager* config1 =
ConfigurationManager::getInstance();
    ConfigurationManager* config2 =
ConfigurationManager::getInstance();

    std::cout << config1->getConfigValue("Setting 1") <<
std::endl;
    std::cout << config2->getConfigValue("Setting 2") <<
std::endl;
}
```