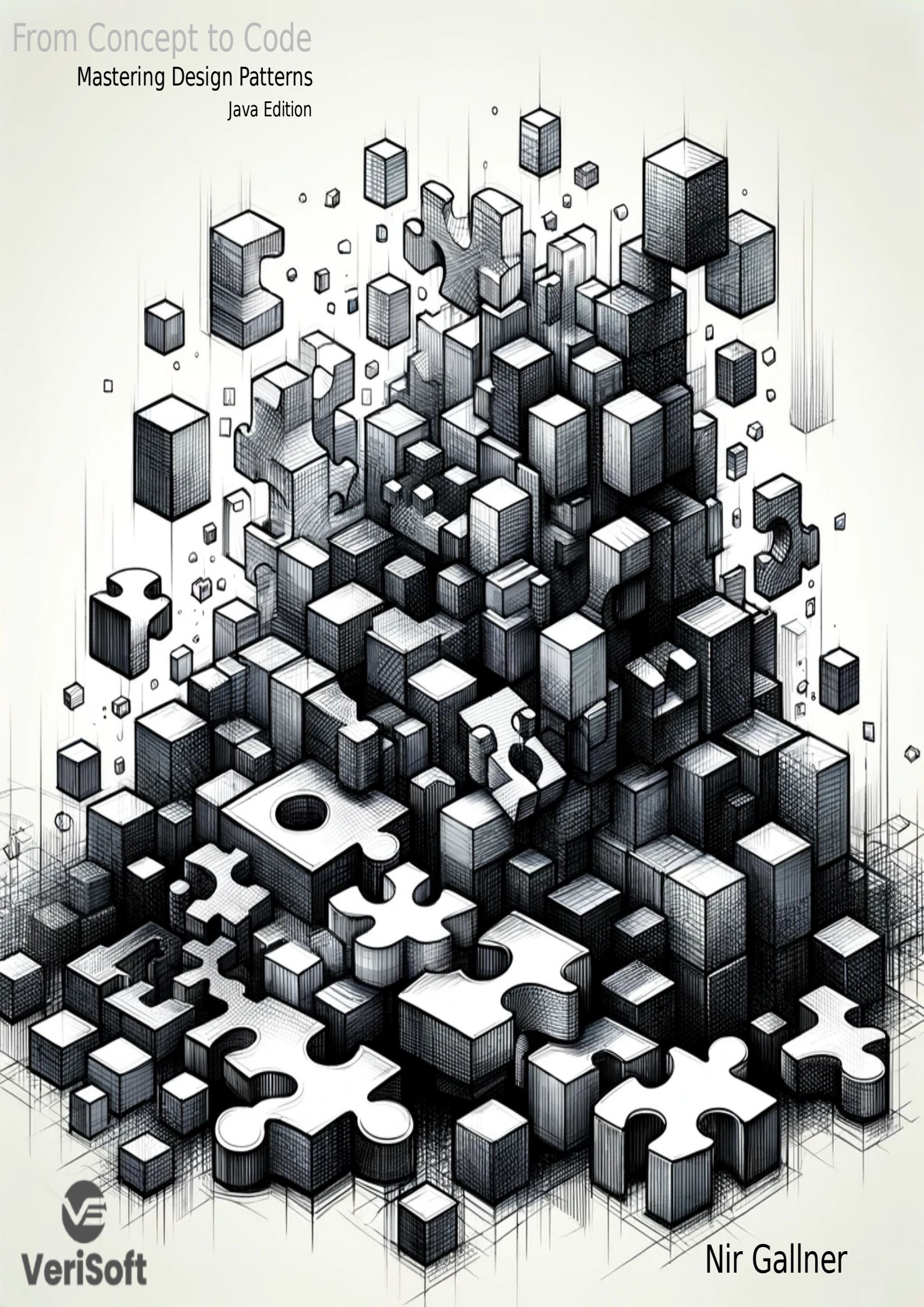
From Concept to Code

Mastering Design Patterns

Java Edition



VeriSoft

Nir Gallner

Chapter 1: The Singleton Pattern

Introduction

Imagine your town decides to have only one library, the central hub for all book lovers. This library is special because it's the only place where everyone can go to borrow books. If you want a book, you go to this library, and if someone else needs the same book, they also visit the same library. There's no option to build another library; this is the one and only. It becomes the go-to spot for everyone's reading needs, ensuring that all book sharing and borrowing happen in a single, managed place.

In the world of software, the Singleton Pattern is like having that one library in town. It ensures that a particular class in a program can be instantiated once and only once.

This unique instance then becomes the central point of access for the specified service or resource throughout the application. Like the town's single library, the Singleton Pattern provides a single, global reference point to the resource or service it represents, making sure that everyone goes through it to access the capabilities it offers.

Key Components

- *Unique Instance*: The Singleton Pattern ensures that only one instance of a particular class can exist within the program, mimicking the one and only library in the town scenario.
- *Central Point of Access*: Similar to how the town's single library serves as the central hub for all book-related activities, the Singleton Pattern designates the instantiated object as the sole access point for the specified service or resource in the application.
- *Global Reference Point*: Just as the town's library serves as the go-to spot for all book-related needs, the Singleton Pattern offers a single, global reference to the instantiated object, ensuring that all interactions with the service or resource go through it.
- *Managed Place*: Like the centralized management of the town's library for book borrowing and sharing, the Singleton Pattern facilitates controlled access and usage of the instantiated object, promoting consistency and coordination within the application.

UML Diagrams

Next, we will explain the concept of the Singleton design pattern using UML.

Class Diagram

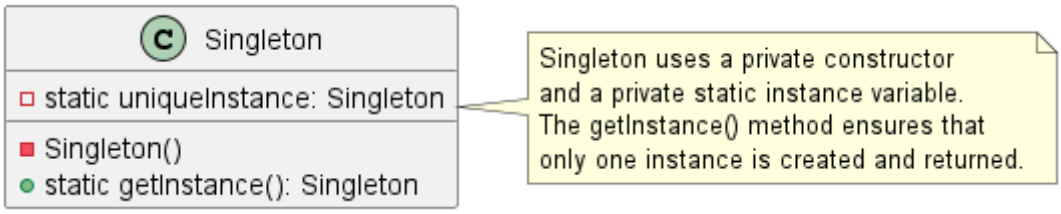


Figure 1. The Singleton Class Diagram

In this class diagram, the `Singleton` class represents the central library in a town. Just like the library, the `Singleton` class ensures that only one instance of itself exists throughout the application. The private constructor and static instance variable function similarly to the library's unique status and limited physical presence, ensuring that no additional instances can be created. The getInstance() method serves as the main entrance to the library, allowing access to the `Singleton` instance and ensuring that all requests for the `Singleton` class are directed to the same instance, analogous to how all book borrowing and sharing in the town are managed through the central library.

Sequence Diagram

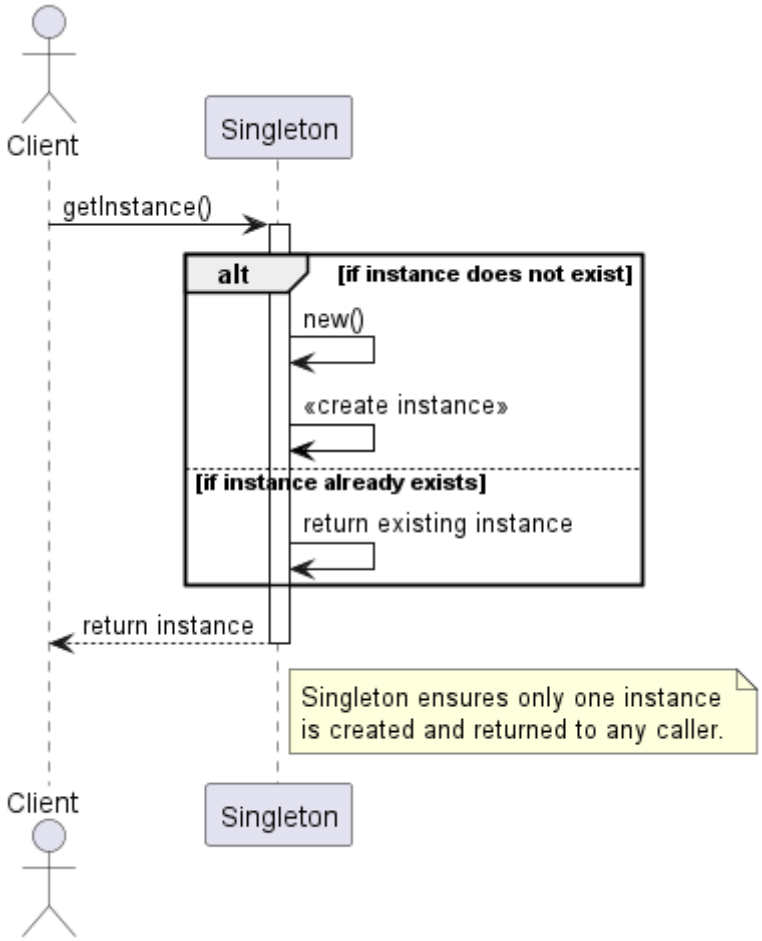


Figure 2. The Singleton Sequence Diagram

In the sequence diagram, the interaction between the Client and `Singleton` classes is illustrated, analogous to a person visiting the town's central library. When the Client sends a request to the `Singleton` class via the getInstance() method, it activates the `Singleton` class, symbolizing someone entering the library. If no instance

of the `Singleton` class exists, represented by the absence of a library patron, the `Singleton` class creates a new instance, akin to a person entering the library for the first time and becoming its first visitor. However, if an instance already exists, indicated by the presence of a library visitor, the `Singleton` class simply returns the existing instance, mirroring how a person returning to the library is not issued a new library card but instead continues using their existing one. Finally, the `Singleton` class sends the instance back to the Client, symbolizing the library providing access to its resources.

Implementation Walkthrough

In this example, we'll implement the Singleton pattern using the library analogy.

The Library class (the Singleton)

```
class Library {
    private static Library uniqueInstance;
    private Library() {}
    public static synchronized Library getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Library();
        }
        System.out.println("Granting access to the central library");
        return uniqueInstance;
    }

    public void borrowABook() {
        System.out.println("Borrowing a book from the library...");
    }
}
```

- The `Library` class represents the central library.
- It contains a private static variable `uniqueInstance` to hold the single instance of the class.
- The constructor is private, ensuring that no other class can instantiate a `Library` object.
- The `getInstance()` method is a static method that returns the single instance of the `Library` class. It creates a new instance if one doesn't exist, otherwise, it returns the existing instance.

Usage Example

```
class Client {
    public static void main(String[] args) {
        Library library = Library.getInstance();

        // Use the library instance
        library.borrowABook();
```

```
    }
}
```

- In the example, it obtains an instance of the `Library` class using the `getInstance()` method.
- After obtaining the object, it uses the `borrowABook()` method to use the `library` object.

Code Output

The above code output is:

```
Granting access to the central library
Borrowing a book from the library...
```

Design Considerations

When implementing the Singleton pattern, several considerations should be taken into account:

- **Thread Safety**: If multiple threads may access the `getInstance()` method simultaneously, ensure thread safety to prevent race conditions. This can be achieved by using synchronization or by utilizing the double-checked locking pattern.
- **Lazy Initialization**: Decide whether the Singleton instance should be lazily initialized (created only when requested) or eagerly initialized (created at application startup). Lazy initialization saves memory by only creating the instance when needed, but it may introduce overhead due to synchronization. In this chapter's code example, the lazy initialization approach was used.
- **Serialization**: If the Singleton class needs to be serialized, ensure that the deserialization process does not create new instances, potentially violating the Singleton pattern. This can be achieved by implementing the `readResolve()` method to return the existing instance during deserialization.
- **Testing**: Test the Singleton class thoroughly to ensure that it behaves as expected in different scenarios, including concurrency testing to verify thread safety.
- **Dependency Injection**: Consider using dependency injection frameworks to manage Singleton instances, especially in larger applications where manual instantiation may lead to tight coupling and decreased maintainability.

By carefully considering these aspects during the design and implementation of the Singleton pattern, developers can create robust and efficient singleton classes that meet the requirements of their applications.

Conclusion

The Singleton pattern provides a simple and effective way to ensure that a class has only one instance throughout the application. By centralizing access to resources or services, it promotes consistency, efficiency, and ease of maintenance. However, it's important to carefully consider design considerations such as thread safety, lazy initialization, serialization, testing, and dependency injection to create a robust Singleton

implementation. When used judiciously and in alignment with the application's requirements, the Singleton pattern can greatly enhance the design and architecture of software systems.

Let's talk about design patterns - those handy tools that makes coding life a breaze. Whether you're just starting out or you're been around the block, they're like secret tricks for writing top-notch software. This book will guide you through it, step by step.

Nir Gallner, with over 20 years of experience in test automation, is the founder and CEO of VeriSoft Testing Services, a boutique company specializing in AI-based software testing.

VeriSoft