Universidad Tecnológica de Tijuana

PATRONES DE DISEÑO

DISEÑO DE APLICACIONES

RODRIGUEZ CACHO XIMENA C.

UNIVERSIDAD TECONOLOGICA DE TIJUANA

TABLE OF CONTENTS

Introduction 3
Definition 3
Conclusion 4
References 5

Introduction

In software development, design patterns are proven and recurring solutions to address common problems in system design. These patterns offer reusable and efficient approaches to solving challenges that developers frequently encounter. By following these patterns, development teams can improve the maintainability, scalability, and flexibility of their applications.

Some key examples of design patterns include the Singleton, which guarantees a single instance of a class; the Factory Method, which delegates the creation of objects to its subclasses; and the Observer, which establishes a one-to-many dependency between objects to report state changes. Other important patterns are the Decorator, which allows you to dynamically add responsibilities to an object; the Strategy, which defines families of interchangeable algorithms; and the Command, which encapsulates requests as objects.

Choosing the appropriate design pattern depends on the specific problem being addressed in the software design. Each pattern has its own advantages and disadvantages, and its proper application can improve the modularity and overall structure of the code. These patterns are valuable tools for developers, providing proven solutions to common challenges in software development.

Definition

Design patterns are proven, recurring solutions to common problems in software design. These patterns offer a reusable and efficient approach to solving problems that software developers frequently face. Below I provide information on some common design patterns:

1**. Singleton:**

- Purpose: Ensure that a class has only one instance and provide a global access point to that instance.

- Example: Implement a single log for event tracking in an application.

2. **Factory Method:**

- Purpose: Define an interface to create an object, but delegate the choice of its type to subclasses, thus creating a particular instance of the class.

- Example: Create different types of documents (PDF, Word, etc.) through a common interface.

3. **Observer:**

- Purpose: Define a one-to-many dependency between objects, so that when an object changes state, all its dependents are automatically notified and updated.

- Example: Implement a notification system where various UI components are updated when the state of an object changes.

4. **Decorator:**

- Purpose: Dynamically attach additional responsibilities to an object.

- Example: Extend the capabilities of an object (such as a viewport) without altering its structure.

5. **Strategy:**

- Purpose: Define a family of algorithms, encapsulate each of them and make them interchangeable. It allows the algorithm to vary independently of the clients that use it.

- Example: Implement different sorting strategies in an application, allowing you to easily switch between them.

6. **Command:**

- Purpose: Encapsulate a request as an object, allowing clients to be parameterized with operations, queue requests, and perform reversible operations.

- Example: Implement a menu system where menu items are objects that encapsulate executable actions.

7. **Adapter:**

- Purpose: Allow incompatible interfaces to work together.

- Example: Adapt an existing class to comply with a specific interface required by the client.

These are just a few examples and many other design patterns exist. The choice of pattern will depend on the specific problem you are addressing in your software design. Each pattern has its advantages and disadvantages, and it is important to select the most suitable one for the specific situation.

Conclusion

In conclusion, design patterns offer an invaluable set of proven solutions to recurring challenges in software design. By adopting these patterns, developers can improve the quality, maintainability, and flexibility of their systems. Proper application of patterns such as Singleton, Factory Method, Observer, Decorator, Strategy, and Command allows development teams to create more robust and modular architectures.

Understanding and effectively using design patterns not only simplifies the development process, but also promotes code reuse and clarity in software structure. By selecting the right pattern for a specific situation, developers can effectively address particular problems and challenges that arise during software development.

In summary, incorporating design patterns into software development practice contributes significantly to building robust and adaptable systems, allowing teams to efficiently address challenges and foster the continuous evolution of their applications.

References

"Patrones de diseño / Design patterns". Refactoring and Design Patterns. Accedido el 5 de febrero de 2024. [En línea]. Disponible: https://refactoring.guru/es/design-patterns

Falcon. "Los 7 patrones de diseño de software más importantes". DEV Community. Accedido el 5 de febrero de 2024. [En línea]. Disponible: https://dev.to/gelopfalcon/los-7-patrones-de-diseno-de-software-mas-importantes-28l2

"Qué son los Patrones de Diseño de software / Design Patterns". Profile Software Services. Accedido el 5 de febrero de 2024. [En línea]. Disponible: https://profile.es/blog/patrones-de-diseno-de-software/