# 6. Design Patterns: Elements of Reusable Object-Oriented Software

by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Audio (MP3) version: https://books.kim/mp3/book/www.books.kim_735_summary-6__Design_Patterns__.mp3

**Summary:**

Design Patterns: Elements of Reusable Object-Oriented Software is a book written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. It was first published in 1994 and has since become one of the most influential books on software engineering. The book provides an introduction to object-oriented design patterns and how they can be used to create reusable software components.

The book begins with an overview of object-oriented programming (OOP) concepts such as classes, objects, inheritance, polymorphism and encapsulation. It then introduces the concept of design patterns which are recurring solutions to common problems encountered when designing OOP systems. The authors provide 23 different design patterns that have been identified from real world projects including Creational Patterns (Abstract Factory, Builder), Structural Patterns (Adapter, Bridge), Behavioral Patterns (Command, Interpreter) and Concurrency Patterns (Monitor). Each pattern is described in detail along with examples showing how it can be applied.

The second part of the book focuses on applying these patterns in practice. This includes topics such as refactoring existing code for better maintainability; using frameworks to reduce development time; testing strategies for ensuring quality; debugging techniques; performance optimization tips; version control best practices; documentation guidelines etc.

Finally the authors discuss some advanced topics related to OOP such as distributed computing architectures like CORBA or COM/DCOM; component based development approaches like JavaBeans or Enterprise JavaBeans etc.; web services technologies like SOAP or WSDL etc.

Overall Design Patterns: Elements of Reusable Object-Oriented Software provides a comprehensive guide for developers looking to use object oriented principles effectively when creating software applications.</

**Main ideas:**

***#1. Strategy Pattern: The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. This allows the algorithm to be selected at runtime depending on the situation. It also allows for the algorithms to be easily changed without affecting the client code.***

The Strategy Pattern is a powerful tool for designing flexible and extensible software. It allows developers to define a family of algorithms, encapsulate each one, and make them interchangeable. This makes it possible to select the appropriate algorithm at runtime depending on the situation. Furthermore, this pattern also enables easy changes in algorithms without affecting the client code.

The Strategy Pattern can be used in many different scenarios where an application needs to dynamically switch between different behaviors or operations based on user input or other conditions. For example, if an application needs to sort data differently depending on user preferences or data type, then using the Strategy Pattern would allow for easily swapping out sorting algorithms as needed.

Overall, the Strategy Pattern provides a great way of creating flexible and extensible software that can adapt quickly to changing requirements. By defining multiple strategies within an application and making them interchangeable at runtime, developers are able to create applications that are more robust and easier to maintain over time.

***#2. Observer Pattern: The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically. This allows for a loosely coupled system where objects can interact without being tightly coupled.***

The Observer Pattern is a powerful tool for creating loosely coupled systems. It defines a one-to-many dependency between objects, so that when one object changes state, all of its dependents are notified and updated automatically. This allows for an efficient way to keep multiple objects in sync without having them tightly coupled.

In the Observer Pattern, there is typically one subject object which other dependent objects observe. When the subject object changes state, it notifies all of its observers who then update themselves accordingly. The observers can also register and unregister with the subject as needed.

The Observer Pattern is useful in many different scenarios such as user interfaces where multiple views need to be kept up to date with each other or when data needs to be shared across multiple components without tight coupling.

***#3. Decorator Pattern: The Decorator Pattern attaches additional responsibilities to an object dynamically. This allows for the responsibilities of an object to be modified without affecting the other objects in the system.***

The Decorator Pattern is a powerful tool for adding functionality to an object without having to modify the underlying code. It allows developers to add new responsibilities and features to existing objects in a flexible way, while still preserving the original objects interface. This pattern can be used when there is a need for additional behavior or state that must be added dynamically at runtime.

The Decorator Pattern works by wrapping an existing class with another class which adds extra functionality. The wrapper class contains references to the original object and delegates all method calls back to it, while also providing its own implementation of certain methods if needed. This allows developers to easily extend the capabilities of an existing object without having to rewrite any of its code.

The Decorator Pattern provides many advantages over traditional inheritance-based approaches such as increased flexibility and reusability. By using this pattern, developers are able to quickly add new features or behaviors without needing extensive modifications or changes in their codebase.

***#4. Factory Method Pattern: The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. This allows for the creation of objects to be delegated to subclasses, which can be easily changed without affecting the client code.***

The Factory Method Pattern is a creational design pattern that provides an interface for creating objects, while allowing subclasses to decide which class to instantiate. This allows the client code to be easily changed without affecting the object creation process. By delegating the responsibility of object creation to subclasses, this pattern helps reduce coupling between classes and makes it easier to modify or extend existing code.

This pattern can be used in situations where there are multiple implementations of a particular type of object that need to be created. For example, if you have different types of cars that need to be created based on user input, then using the Factory Method Pattern would allow you create any type of car without having to write separate code for each one. The factory method also allows for easy extensibility as new types of cars can easily be added by simply adding a new subclass.

***#5. Singleton Pattern: The Singleton Pattern ensures that a class has only one instance and provides a global point of access to it. This allows for a single instance of a class to be shared across the system, which can be easily accessed and modified.***

The Singleton Pattern is a powerful design pattern that ensures only one instance of a class can exist at any given time.

This allows for the single instance to be shared across the system, providing an easy way to access and modify it. The Singleton Pattern also provides a global point of access to this single instance, allowing other classes or objects in the system to easily interact with it.

Using the Singleton Pattern helps ensure consistency throughout your application by ensuring that all parts of your code are using the same object. It also simplifies memory management as there is no need to create multiple instances of an object when only one will suffice. Additionally, since there is only ever one instance of a class available, you can more easily control how data flows through your application.

When implementing the Singleton Pattern in your codebase, its important to consider thread safety and performance implications. If multiple threads attempt to access or modify the same singleton instance simultaneously, race conditions may occur which could lead to unexpected results or errors. To prevent this from happening, proper synchronization techniques should be used when accessing and modifying singletons.

***#6. Command Pattern: The Command Pattern encapsulates a request as an object, allowing for the parameters of the request to be easily changed. This allows for requests to be queued or logged, and for the undoing and redoing of operations.***

The Command Pattern is a powerful tool for creating flexible and extensible applications. It allows requests to be encapsulated as objects, allowing the parameters of the request to be easily changed. This makes it possible to queue or log requests, and also enables undoing and redoing operations.

Using the Command Pattern can help reduce code complexity by separating out different parts of an application into distinct objects that are responsible for handling specific tasks. This helps keep code organized and maintainable, making it easier to debug problems or add new features in the future.

The Command Pattern also provides a way to decouple components from each other, which can make applications more robust when changes need to be made. By using commands instead of direct calls between components, developers can ensure that any changes they make will not affect other parts of their system.

***#7. Adapter Pattern: The Adapter Pattern converts the interface of a class into another interface that the client expects. This allows for classes with incompatible interfaces to work together, and for existing classes to be reused in new systems.***

The Adapter Pattern is a powerful tool for creating flexible and reusable code. It allows developers to create classes with incompatible interfaces that can still work together, as well as allowing existing classes to be reused in new systems. This pattern works by converting the interface of one class into another interface that the client expects. The adapter acts as a bridge between two different objects, allowing them to communicate without having to modify either object.

This pattern is especially useful when dealing with legacy code or third-party libraries which may have an outdated or incompatible interface. By using an adapter, developers can easily integrate these components into their own system without needing to rewrite any of the existing code.

The Adapter Pattern also helps reduce coupling between components by providing a layer of abstraction between them. This makes it easier for developers to make changes and updates without affecting other parts of the system.

***#8. Facade Pattern: The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. This allows for a simpler interface to be provided to the client, which hides the complexity of the subsystem.***

The Facade Pattern is a design pattern that provides a unified interface to a set of interfaces in a subsystem. This allows for the client to interact with the subsystem through one simplified interface, rather than having to deal with the complexity of each individual component within it. By providing this single point of access, clients can more easily use

and understand the functionality provided by the subsystem.

The Facade Pattern also helps reduce coupling between components within the system. By hiding all but one interface from view, clients are not exposed to any unnecessary details about how those components work together internally. This reduces dependencies between different parts of the system and makes it easier for developers to make changes without affecting other areas.

Finally, using this pattern can help improve performance as well since only one call needs to be made instead of multiple calls when dealing with complex systems. This simplifies code and makes it easier for developers to maintain their applications over time.

***#9. Template Method Pattern: The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. This allows for the common parts of an algorithm to be defined in a single method, while allowing subclasses to provide the implementation for the steps that vary.***

The Template Method Pattern is a powerful tool for creating algorithms that can be easily adapted to different situations. It defines the skeleton of an algorithm in a single method, allowing subclasses to provide the implementation for steps that vary. This allows developers to create algorithms with common parts defined in one place, while still providing flexibility and customization through subclass implementations.

Using this pattern helps reduce code duplication and makes it easier to maintain complex algorithms. By defining the structure of an algorithm in one place, developers can quickly make changes or add new features without having to rewrite large sections of code. Additionally, by separating out the varying parts into separate classes, it becomes much simpler to test each part individually.

Overall, the Template Method Pattern provides a great way for developers to create flexible and extensible algorithms that are easy to maintain over time.

***#10. Iterator Pattern: The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. This allows for the elements of an aggregate object to be accessed in a consistent manner, without the client needing to know the underlying structure of the object.***

The Iterator Pattern is a powerful tool for accessing the elements of an aggregate object in a consistent manner. It allows clients to access the elements of an aggregate object without needing to know its underlying structure. This makes it easier for clients to traverse and manipulate collections of objects, as they can do so without having to understand how the collection is structured.

The pattern also provides flexibility when dealing with different types of collections. By using iterators, clients can easily switch between different implementations or data structures while still being able to access their contents in a uniform way. This means that code written using iterators will be more reusable and maintainable than code which relies on specific implementations.

Finally, by abstracting away the details of how an aggregate object is represented internally, iterator patterns make it easier for developers to modify existing code or add new features without breaking existing functionality. This helps ensure that applications remain robust and reliable even after changes have been made.